SYNTHESIS CONSTRAINT OPTIMIZATION FOR NEAR-THRESHOLD VOLTAGE DESIGN

BY

SEUNG WON MIN

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2017

Urbana, Illinois

Adviser:

Associate Professor Nam Sung Kim

Abstract

Near threshold voltage (NTV) design is an optimal design choice for IoT devices due to its low energy consumption, but it is at the same time vulnerable to process variation. Under severe process variation, IC designers should choose to lower the yield rate and keep the performance or to lower the operating frequency and keep the yield rate. To design more predictable devices, IC designers can intentionally increase critical path depths, remove certain cells which are susceptible to process variation from the synthesis library, or replace them with larger devices. These solutions, however, come with a cost. Making critical paths longer decreases device performance and replacing or removing some devices results in higher power consumption. In this project, we apply multiple synthesis constraints with TSMC 65nm PDK on an openMSP430 microcontroller and measure the performance and power consumption in an NTV environment. We find out that removing cells which are susceptible to the process variation during the synthesis process can ultimately negatively impact performance and power characteristics. Using larger cells yields a better performance with smaller process variation, but at the same time it significantly increases the power consumption.

Table of Contents

1. Introduction

For mobile and IoT devices, accomplishing high energy efficiency has become one of the most important design objectives, as the operating time of these mobile and IoT devices is strictly limited by their battery capacity. To achieve both high energy efficiency and reasonable performance, near threshold computing (NTC) has emerged. As plotted in Figure 1, Dreslinski et al. [1] demonstrated that energy consumption per operation super-linearly decreases as the supply voltage decreases from nominal voltage to threshold voltage. Since the delay also increases exponentially as the voltage decreases, we can find a balanced energy-performance trade-off point slightly above the threshold voltage. That is, such devices consume approximately one tenth the energy at near-threshold voltage than at super-threshold voltage with reasonable delay increase (or performance degradation). Considering most mobile and IoT devices do not require high performance, the aforementioned trade-off point is suitable.

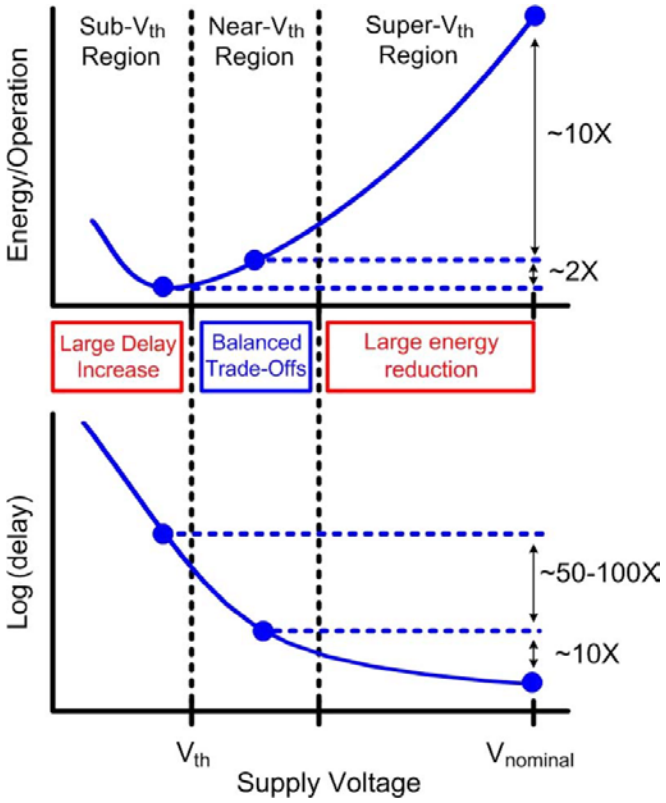


Figure 1. Energy and delay in different supply voltage operating regions *[1]*.

However, relatively high delay is not the only disadvantage that should be considered when designing an NTC device. The impact of random process variation on delay is relatively small at super-threshold voltage, but this becomes a significant problem at near-threshold voltage, as the delay sensitivity to process variation super-linearly increases.

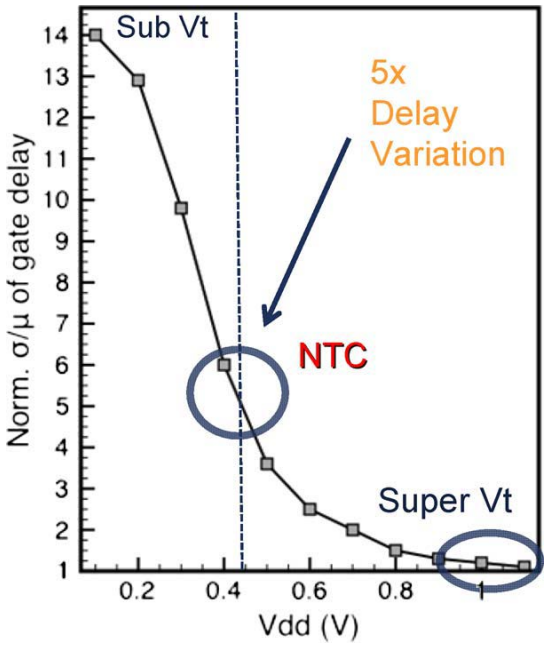


Figure 2. Impact of voltage scaling on gate delay variation *[1]*.

In Figure 2, small voltage swing at super-threshold voltage results in small difference in gate delay, but the same degree of voltage swing at near-threshold voltage leads to large difference in gate delay. The process variation causes each transistor to have different threshold voltage and channel length, the effects of which can be amplified at near-threshold voltage. If IC designers build circuits working at near-threshold voltage, they must consider delay variation due to process variation, as such delay variation may cause critical paths of certain chips to violate given timing constraints. Furthermore, if designs are synthesized by using standard libraries which are not targeted for near-threshold voltage environment, they can have unbalanced delays across different stages, as the delays of different gates scales differently as the supply voltage scales down [2]. However,

the process variation can also be exploited to improve the performance. Tiwari et al. [3] proposed to implement a clock-skewing module in the registers of each pipeline stage, as delay variation is expected due to the process variation. Whenever the delay variation is detected after a chip is manufactured, the clock-skewing module in each pipeline stage of the chip adjusts the clock arrival time and thus the timing constraint of each stage.

Instead of skewing clock signals for each pipeline stage, we can attempt to change the delay of each pipeline stage by controlling the supply voltage. If a certain stage has shorter critical path delay than other stages, it can be slowed down by reducing the supply voltage. In contrast, we can attempt to increase the voltage to mitigate the delay variation. With the voltage controlling mechanism, we can reduce the power consumption while maintaining the performance. In this project, we focus on within-die process variations, which can be largely divided into random correlated and uncorrelated process variations, and make the following key contributions:

1. We characterized the delay variation of two microprocessor designs caused by process variation in near-threshold voltage environment, and demonstrated that within-die random uncorrelated process variation can significantly affect the delay of critical paths even in 65nm technology generation, and thus energy efficiency when we attempt to satisfy the timing constraint.
2. We revealed that microprocessors for IoT devices are too small to be affected by within-die random correlated process variation. Although we assume that notable within-die random correlated process variation exists and leads to different unbalanced delays across pipeline stages, we cannot cost-effectively adjust the delay of individual pipeline stages using fine-grained LDOs. This is because different pipeline stages often share some resources and are inter-connected in realistic processor designs. That is, unlike typical microprocessors illustrated in computer

architecture textbooks, the pipeline stages are not totally independent and isolated by the pipeline registers.

3. We proposed various logic synthesis techniques including:
    a. Limiting the use of gates with a large number of fan-ins such as 4-input NAND or NOR gates, as they require stacking 4 transistors serially, weakening the overall equivalent transistor and thus making it more sensitive to process variations.
    b. Limiting the use of gates with minimum or small transistor sizes on the critical paths, as gates with small transistors are more sensitive to random process variation.
    c. Characterizing the delay sensitivity of all the gates to process variations and selectively and gradually replacing gates that are most sensitive to process variations.
    d. Targeting higher frequency than needed to use gates with larger cells.

After extensively applying these four techniques to make designs more robust to process variations, we make the following observations. The designs with these four techniques become less sensitive to process variations and exhibit less delay variation. For example, our experiment shows that the designs synthesized with gates with the smallest transistor become very sensitive to process variations. A design sample at the 3 sigma point exhibits 16% frequency degradation at the near-threshold voltage regime. In contrast, design samples with these four techniques at the 3 sigma point exhibit 4-6% frequency degradation. However, designs with these four techniques consume higher dynamic and leakage power at given voltage.

For fair comparisons between these designs, we scale up the voltage of all the aforementioned designs such that the 3 sigma samples of these designs operate at the same frequency. When we compare the energy consumption of all these designs at

the same frequency, we discover that it is always better to synthesize a design with gates with the smallest transistor and then scale the voltage of the design up to counter the negative effect of process variation at near-threshold voltage regime.

The reason for this behavior is as follows. In accomplishing the same target frequency, the designs with smaller transistors need higher voltage, but the voltage increase to compensate the delay variations is small, as the delay changes very dramatically at the near-threshold voltage regime (Figure 2). Consequently, the power increase associated with the process variation is small, as well.

4. We find the best synthesis constraints which can achieve the best energy efficiency and the smallest delay variation at the near-threshold regime, combined with a global voltage scaling technique. That is, gates with the smallest transistor are sufficient to make a design operate at a target frequency appropriate for IoT applications and consume the least power even considering the post-silicon voltage scaling.

2. Experiment Methodology

This chapter describes the general simulation set-up and methodology used for experiments. To observe the effect of random variation over logic design, we use TSMC 65nm GP (General Purpose) standard cell library. This library is capable of simulating MOSFET device mismatch with SPICE or Spectre simulator. Based on the library, we use Synopsys Design Compiler (DC) to generate a gate-level Verilog code of a given logic design. To get dynamic power consumption of the design, we follow a method explained in Figure 3. First, wrap the Verilog code with testbench and generate multiple possible input vectors for the testing. Next, we input the test setup into Synopsys Verilog Compiler Simulator (VCS) to obtain a Value Change Dump (VCD) file which contains the switching activity of each net in the design. This VCD file can be fed into Synopsys PrimeTime (PT) to calculate an accurate dynamic power consumption. Instead of using VCD, we can use SAIF or Vector-free method (Table 1). Since the duration of simulation is short and the average size of netlist small in our experiments, we will continue using VCD flow.

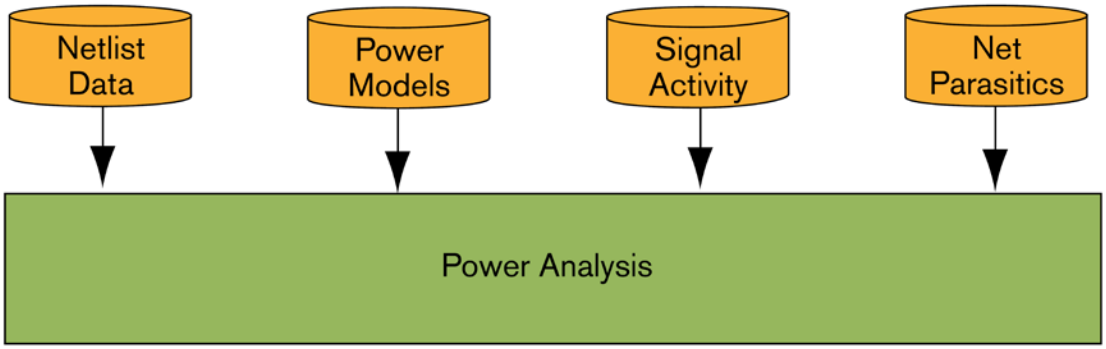
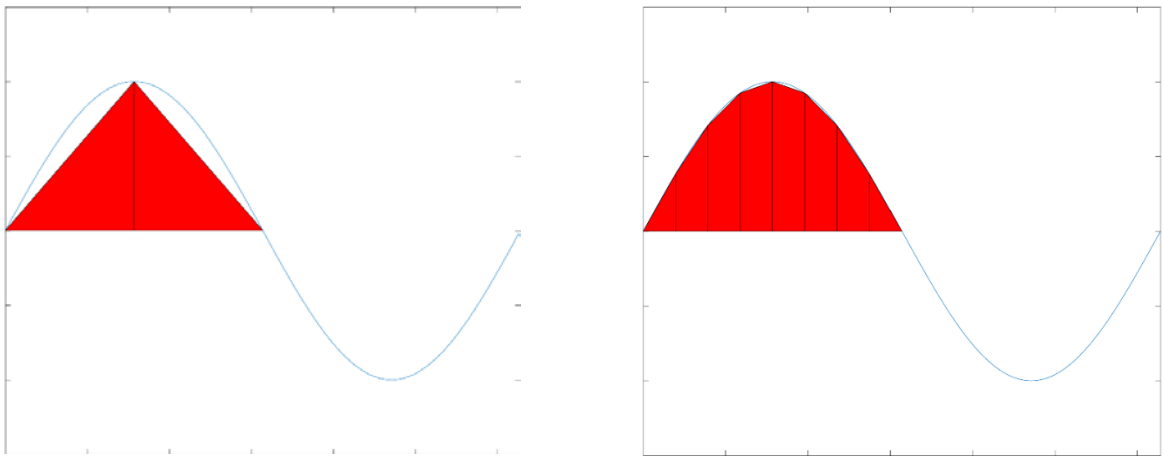


Figure 3. Power analysis requirements *[4]*.

Table 1. Power analysis flow

Flow	Inputs			Outputs
	Power Models	Netlist & Parasitics	Signal Activity Type	
VCD	Required	Required	Time-Accurate	Peak power and Average power
SAIF	Required	Required	Averaged	Average power
Vector-Free	Required	Required	None	Average power

However, we can only calculate the power and the delay based on the input library, which has 1.0 V of fixed VDD value. To evaluate the circuit under NTV environment, we additionally generate a SPICE Netlist of the circuit. The conversion of the gate-level Verilog to the SPICE Netlist can be easily done by using Nettran command of Synopsys IC Validator (ICV). This command does one-to-one mapping of Verilog gate instances to SPICE instances. With the output of this command, we can do DC analysis with Synopsys HSPICE to find the leakage current. We assume the leakage power of the circuit is equal to the multiple of the input voltage and the measured leakage current. As we lower the voltage further, there is a possibility of divergence while running simulations. To prevent this problem, we force the simulator to use the GEAR integration method instead of regular trapezoidal integration. The concept of trapezoidal integration is shown in Figure 4. In every step, we select two points of the function and draw a line. Once we reach an endpoint, we add the areas of all trapezoids (or triangles). By comparing Figure 4 (a) and (b), we can see the accuracy depends heavily on the number of steps we used to calculate the integration. This simplicity is beneficial for the shorter computation time, but if the step size is not small enough, it may not converge and so may cause oscillation in some calculations. Even worse, it can converge to a wrong value and finish a simulation without reporting any error. The GEAR method used in HSPICE is based on GEAR [5]. The method uses a weighted average of past steps to determine the next step. This difference can help GEAR to converge while the trapezoidal method fails, but it also adds an additional complexity which increases the overall simulation time. Unfortunately, we cannot predict if any simulations will fail to converge before they are finished. The overhead of identifying simulations which failed to converge and re-launching them with GEAR method is not negligible, so we decide to use GEAR by default.



(a) (b)

Figure 4. Comparison of trapezoidal rule integration methods.

Dynamic power can also be gathered by using the SPICE Netlist, but the simulation can take a significant amount of time before all test vectors are consumed. Therefore, we estimate the dynamic power based on the knowledge that $P_{dyn} = f \times VDD^2 \times C$. From this equation, we can assume

$$\frac{P_{new}}{P_{baseline}} = \frac{f_{baseline}}{f_{new} \times VDD_{new}^2} (VDD_{baseline} = 1) \qquad \text{(eq. 1)}$$

Here, $P_{baseline}$ is a value obtained from PT.

Finding the delay of the critical path requires another command, write_spice_deck, from PT. The reason for using another command is that DC and PT use static timing analysis (STA). STA can be done quickly since it simply adds the gate delays from the input to the output, but it is ignorant of when the propagation can happen. For example, say DC reports the critical path of a 2-input AND gate is occurring when input A rises and output Y also rises. In such a simple case, it can be easily noticed that the other input B must stay high to make that transition true. However, as the logic becomes more and more complicated, it is hard for IC designers to figure out what the other inputs should be to trigger the critical path. Fortunately, write_spice_deck can generate a SPICE Netlist and stimulus for just the critical path portion. The command automatically sets the side input

values in SPICE code so we only need to change the input voltage and run with HSPICE. Figure 5 shows the part of the critical path reported by PT. After running write_spice_deck command, PT outputs a SPICE Netlist shown in Figure 6. At 1.0 V of VDD, PT timing report and the SPICE Netlist simulation give very similar values; thus, we can safely assume the command is generating a correct SPICE Netlist. When we measure circuit delay, we define delay as the time between VDD/2 point of the input signal to VDD/2 point of the output signal. For example, if we measure delay of an inverter in Figure 7, we measure the time shown in Figure 8.

Once the SPICE Netlists are ready, we add sweep monte argument to perform Monte Carlo simulation. This argument commands HSPICE to simulate a given SPICE Netlist multiple times with different random seeds, which are used to determine random variations of each MOSFET existing in the SPICE Netlist. To capture the wide range of variation, we run Monte Carlo 200 times for every experiment. Similar to the leakage power analysis, we also use GEAR for the transient analysis.

High level description of overall simulation chain is described in Figure 9.

```
Startpoint: exec_cycle_i_reg
            (rising edge-triggered flip-flop clocked by clk)
Endpoint: alu_stat_reg_0_
            (rising edge-triggered flip-flop clocked by clk)
Path Group: clk
Path Type: max

Point                                                    Incr       Path
-----------------------------------------------------------------------------
clock clk (rise edge)                                  0.0000     0.0000
clock network delay (ideal)                            0.0000     0.0000
exec_cycle_i_reg/CK (DFFQX0P5MA10TR)                   0.0000     0.0000 r
exec_cycle_i_reg/Q (DFFQX0P5MA10TR)                    0.1140 *   0.1140 r
alu_0/U42/A (NAND2X0P7BA10TR)                          0.0000 *   0.1140 r
alu_0/U42/Y (NAND2X0P7BA10TR)                          0.0650 *   0.1790 f
alu_0/U43/A (INVX0P5MA10TR)                            0.0000 *   0.1790 f
alu_0/U43/Y (INVX0P5MA10TR)                            0.0810 *   0.2600 r
alu_0/U44/B0 (OAI22X0P5MA10TR)                         0.0000 *   0.2600 r
alu_0/U44/Y (OAI22X0P5MA10TR)                          0.0620 *   0.3220 f
alu_0/U22/A (INVX0P5MA10TR)                            0.0000 *   0.3220 f
alu_0/U22/Y (INVX0P5MA10TR)                            0.0540 *   0.3760 r
alu_0/U87/A0 (AOI22X0P5MA10TR)                         0.0000 *   0.3760 r
alu_0/U87/Y (AOI22X0P5MA10TR)                          0.0530 *   0.4290 f
alu_0/U92/A0 (AOI21X0P5MA10TR)                         0.0000 *   0.4290 f
alu_0/U92/Y (AOI21X0P5MA10TR)                          0.0550 *   0.4840 r
alu_0/U100/B (XOR2X0P5MA10TR)                          0.0000 *   0.4840 r
alu_0/U100/Y (XOR2X0P5MA10TR)                          0.0600 *   0.5440 r
.
.
.
.
.
alu_0/U366/DN (NAND4XXXBX0P5MA10TR)                    0.0000 *   2.0940 f
alu_0/U366/Y (NAND4XXXBX0P5MA10TR)                     0.0720 *   2.1660 f
alu_0/U6/C (NOR3X0P5AA10TR)                            0.0000 *   2.1660 f
alu_0/U6/Y (NOR3X0P5AA10TR)                            0.0570 *   2.2230 r
alu_0/U369/A1 (OAI211X0P5MA10TR)                       0.0000 *   2.2230 r
alu_0/U369/Y (OAI211X0P5MA10TR)                        0.0410 *   2.2640 f
alu_0/U9/C (NOR3X0P5AA10TR)                            0.0000 *   2.2640 f
alu_0/U9/Y (NOR3X0P5AA10TR)                            0.0730 *   2.3370 r
alu_0/U19/B (NAND2X0P5AA10TR)                          0.0000 *   2.3370 r
alu_0/U19/Y (NAND2X0P5AA10TR)                          0.0280 *   2.3650 f
alu_0/U30/B0 (OAI21X0P5MA10TR)                         0.0000 *   2.3650 f
alu_0/U30/Y (OAI21X0P5MA10TR)                          0.0280 *   2.3930 r
alu_0/U377/B0 (AOI22X0P5MA10TR)                        0.0000 *   2.3930 r
alu_0/U377/Y (AOI22X0P5MA10TR)                         0.0280 *   2.4210 f
alu_stat_reg_0_/D (DFFQX0P5MA10TR)                     0.0000 *   2.4210 f
data arrival time                                                 2.4210

clock clk (rise edge)                                 27.3600    27.3600
clock network delay (ideal)                            0.0000    27.3600
clock reconvergence pessimism                          0.0000    27.3600
alu_stat_reg_0_/CK (DFFQX0P5MA10TR)                              27.3600 r
library setup time                                    -0.0260 *  27.3340
data required time                                               27.3340
-----------------------------------------------------------------------------
data required time                                               27.3340
data arrival time                                                -2.4210
-----------------------------------------------------------------------------
slack (MET)                                                      24.9130
```

Figure 5. PT timing report (Critical Path) of openMSP430 ALU.

```
.
.
.
.
* .pin(sub_node) for exec_cycle_i_reg
(DFFQX0P5MA10TR): .Q(exec_cycle_i_reg/Q) .CK(exec_cycle_i_reg/CK) .D(exec_cycle_i_r
eg/D)
xexec_cycle_i_reg exec_cycle_i_reg/Q exec_cycle_i_reg/CK exec_cycle_i_reg/D
DFFQX0P5MA10TR
* .pin(sub_node) for alu_0/U157
(AND2X0P5MA10TR): .Y(alu_0/U157/Y) .A(alu_0/U157/A) .B(alu_0/U157/B)
xalu_0/U157 alu_0/U157/Y alu_0/U157/A alu_0/U157/B AND2X0P5MA10TR
* .pin(sub_node) for alu_0/U37
(AND2X0P5MA10TR): .Y(alu_0/U37/Y) .A(alu_0/U37/A) .B(alu_0/U37/B)
xalu_0/U37 alu_0/U37/Y alu_0/U37/A alu_0/U37/B AND2X0P5MA10TR
* .pin(sub_node) for alu_0/U144
(AOI32X0P5MA10TR): .Y(alu_0/U144/Y) .A0(alu_0/U144/A0) .A1(alu_0/U144/A1) .A2(alu_0
/U144/A2) .B0(alu_0/U144/B0) .B1(alu_0/U144/B1)
xalu_0/U144 alu_0/U144/Y alu_0/U144/A0 alu_0/U144/A1 alu_0/U144/A2
+        alu_0/U144/B0 alu_0/U144/B1 AOI32X0P5MA10TR
.
.
.
.
.
.
(OAI211X0P5MA10TR): .Y(alu_0/U369/Y) .A0(alu_0/U369/A0) .A1(alu_0/U369/A1) .B0(alu_
0/U369/B0) .C0(alu_0/U369/C0)
xalu_0/U369 alu_0/U369/Y alu_0/U369/A0 alu_0/U369/A1 alu_0/U369/B0
+        alu_0/U369/C0 OAI211X0P5MA10TR
* .pin(sub_node) for alu_0/U9
(NOR3X0P5AA10TR): .Y(alu_0/U9/Y) .A(alu_0/U9/A) .B(alu_0/U9/B) .C(alu_0/U9/C)
xalu_0/U9 alu_0/U9/Y alu_0/U9/A alu_0/U9/B alu_0/U9/C NOR3X0P5AA10TR
* .pin(sub_node) for alu_0/U19
(NAND2X0P5AA10TR): .Y(alu_0/U19/Y) .A(alu_0/U19/A) .B(alu_0/U19/B)
xalu_0/U19 alu_0/U19/Y alu_0/U19/A alu_0/U19/B NAND2X0P5AA10TR
* .pin(sub_node) for alu_stat_reg_1_
(DFFQX0P5MA10TR): .Q(alu_stat_reg_1_/Q) .CK(alu_stat_reg_1_/CK) .D(alu_stat_reg_1_/
D)
xalu_stat_reg_1_ alu_stat_reg_1_/Q alu_stat_reg_1_/CK alu_stat_reg_1_/D
DFFQX0P5MA10TR
* .pin(sub_node) for alu_0/U30
(OAI21X0P5MA10TR): .Y(alu_0/U30/Y) .A0(alu_0/U30/A0) .A1(alu_0/U30/A1) .B0(alu_0/U3
0/B0)
xalu_0/U30 alu_0/U30/Y alu_0/U30/A0 alu_0/U30/A1 alu_0/U30/B0 OAI21X0P5MA10TR
* .pin(sub_node) for alu_0/U377_C_SPC18
(AOI22X0P5MA10TR): .Y(alu_0/U377_C_SPC18/Y) .A0(alu_0/U377_C_SPC18/A0) .A1(alu_0/U3
77_C_SPC18/A1) .B0(alu_0/U377_C_SPC18/B0) .B1(alu_0/U377_C_SPC18/B1)
xalu_0/U377_C_SPC18 alu_0/U377_C_SPC18/Y alu_0/U377_C_SPC18/A0
alu_0/U377_C_SPC18/A1 alu_0/U377_C_SPC18/B0
+        alu_0/U377_C_SPC18/B1 AOI22X0P5MA10TR
* .pin(sub_node) for alu_stat_reg_0_
(DFFQX0P5MA10TR): .Q(alu_stat_reg_0_/Q) .CK(alu_stat_reg_0_/CK) .D(alu_stat_reg_0_/
D)
xalu_stat_reg_0_ alu_stat_reg_0_/Q alu_stat_reg_0_/CK alu_stat_reg_0_/D
DFFQX0P5MA10TR
.
.
.
.
```

Figure 6. SPICE Netlist critical path converted by PT

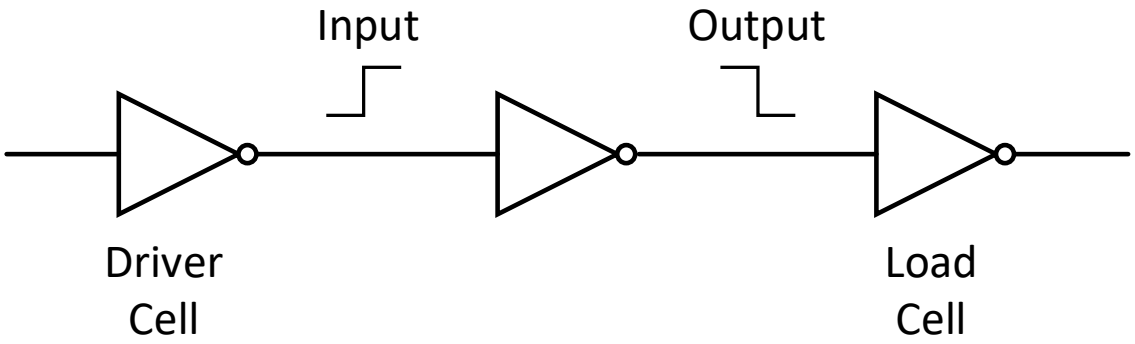


Figure 7. Delay measurement example.

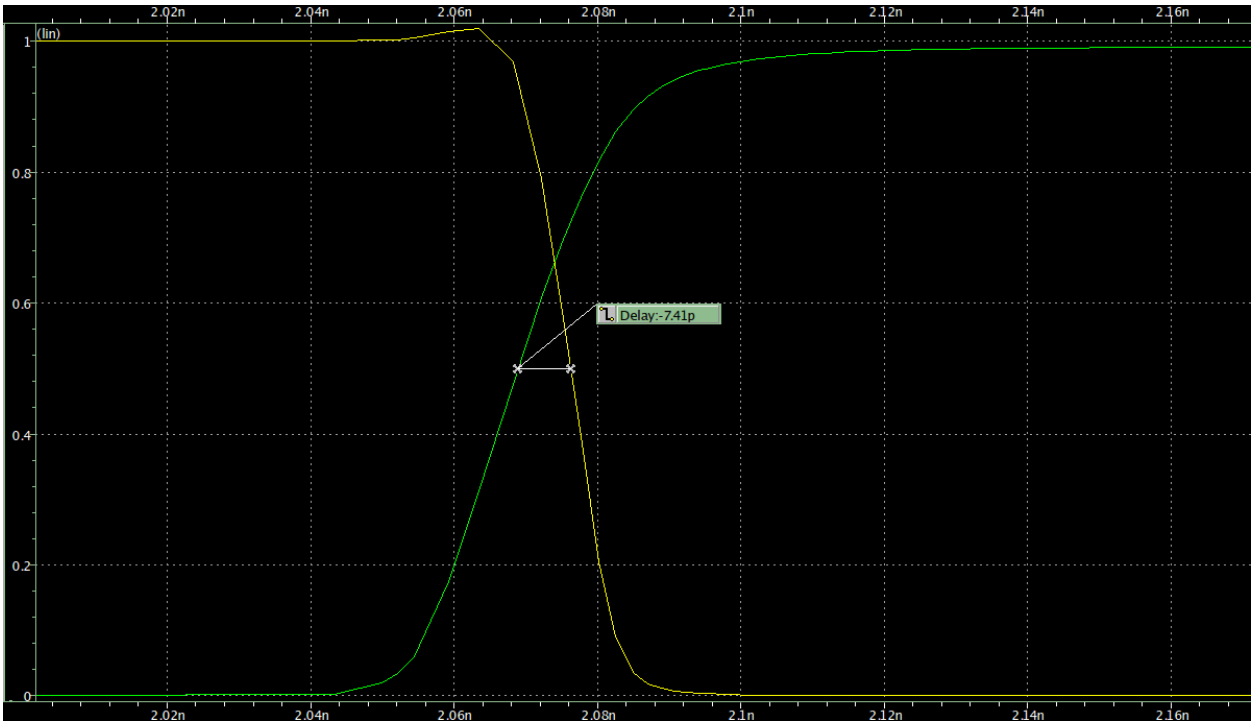


Figure 8. Inverter delay measurement.

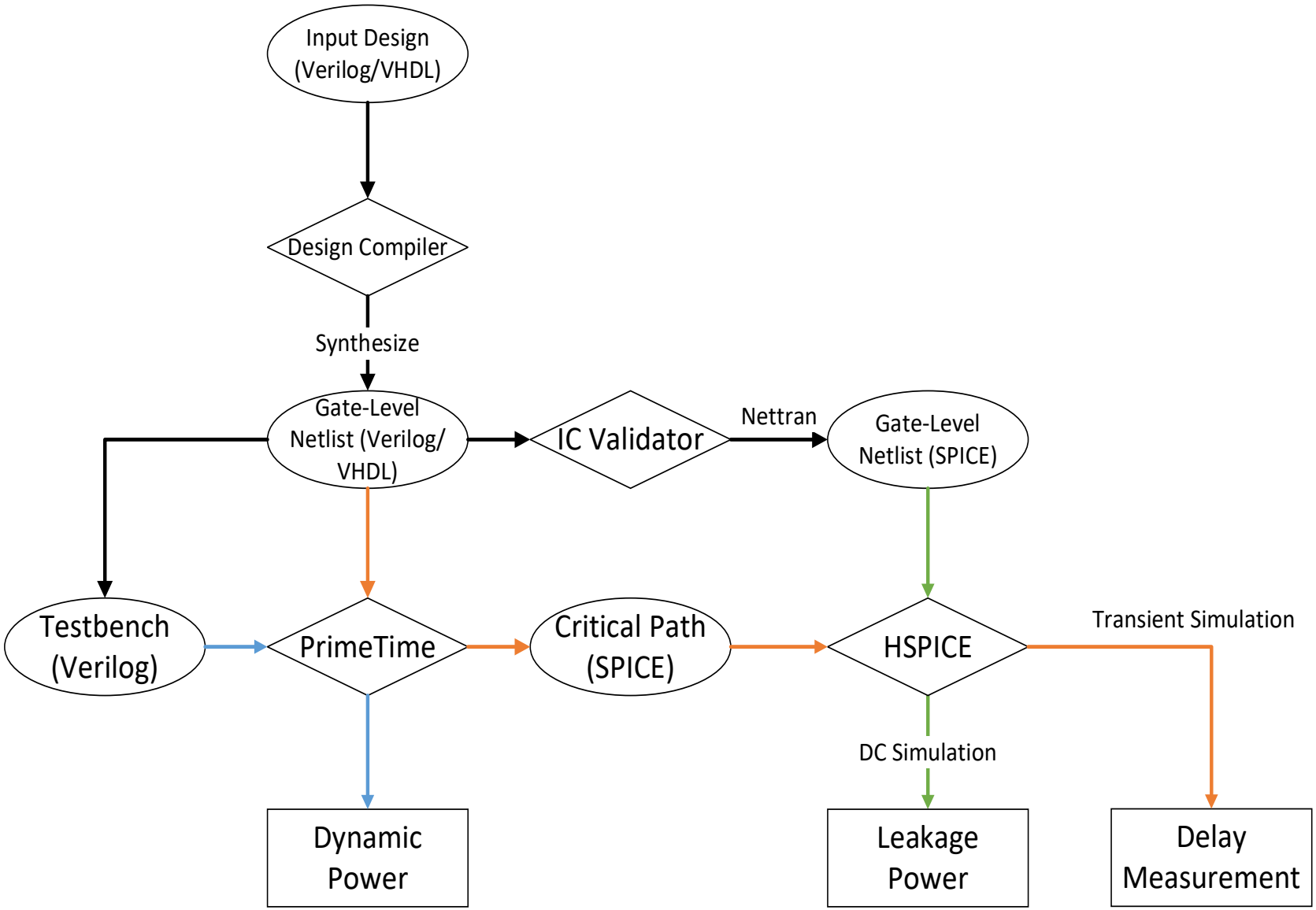


Figure 9: High level description of experiment setup.

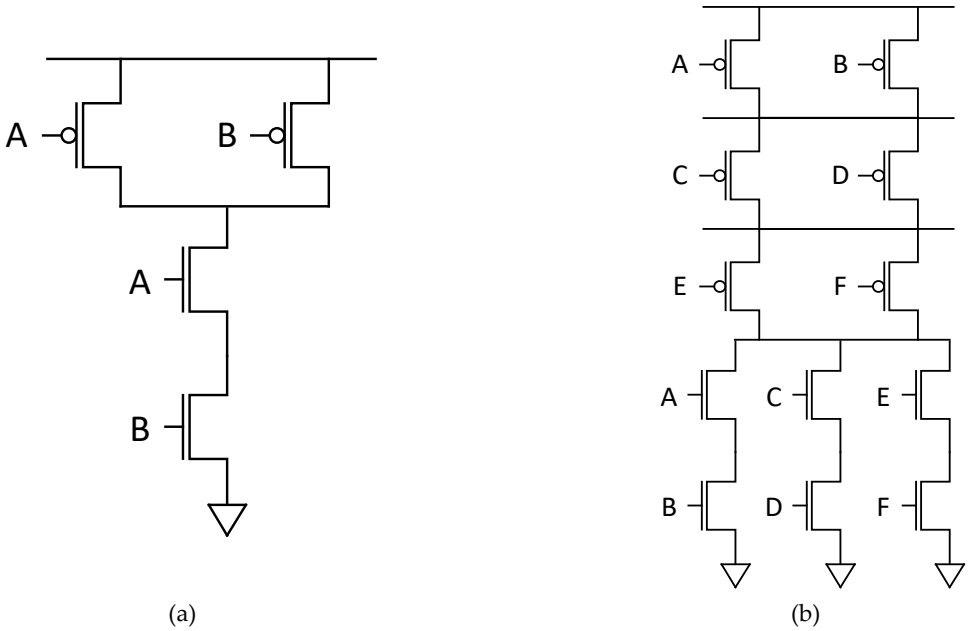
3. Experiments

3.1 Gate Constraint Experiment

In the following set of experiments, we remove a certain set of cells which are vulnerable to the random process variation in NTV environment and study how the removal of the gates can decrease variation of the critical path delay. Without those cells, DC will replace the logic with less sensitive cells.

3.1.1 Large Fan-in Gates

First, we start with removing large fan-in gates. The assumption behind this experiment is that the larger gates tend to include longer chains of transistors. Since the path with a longer chain of transistor results in higher resistance, we assume this path is more susceptible to the random process variation under NTV environment. In Figure 10, NAND2 gate has total 2 inputs and maximum of 2-transistor stack from VDD or GND to the output. On the other hand, AOI222 gate has 6 inputs and maximum of 3-transistor stack from VDD or GND to the output.



(a) (b)

Figure 10. Transistor-level depiction of (a) NAND2 gate and (b) AOI222 gate.

3.1.1.1 Experiment Setup

To test our idea, we categorize cells into three types: 2-input gates, 3-input gates, and 4 or more input gates. Baseline design is using all three types of cells and we start to remove from 4 or more input gates to 3-input gates from the synthesis library. We use openMSP430 microcontroller [6] and Atlas Processor Core [7] for the testing. Both designs are synthesized with maximum possible frequencies, and then we lower VDD to 0.5 V to simulate the NTV environment. For the evaluation of each design, we use mean value of the critical path delay and standard variation of the critical path delay from Monte Carlo simulation. The standard deviation of the delay will show the effect of random process variation. Static and dynamic power numbers are also gathered to evaluate the efficiency of the designs. For the synthesis of both designs, we use tcl script written for openMSP430 microcontroller. Atlas processor core does not have any given synthesis script so we modify openMSP430's script for the synthesis purpose. One thing we must be careful about while synthesizing and analyzing the critical path openMSP430 microcontroller is a false path, which is a path that cannot (or will never) be triggered in reality. Since both DC and PT analyze critical paths based on STA, it is possible they misunderstand false paths as critical paths. A good example of this problem is a carry skip adder (CSA). In Figure 11, the path highlighted in red cannot be a critical path of CSA because if it were the case P logic would already direct the carry value to skip to the end. When DC borrows the adder implementation from Synopsys Designware library, this issue is taken and the path is not considered as the critical path. On the other hand, once the design is synthesized and transformed into gate-level netlist, this information no longer exists. Therefore, when PT directly reads this netlist without any user interference, it considers the path as a critical path. And unfortunately, there are several false paths also in openMSP430 microcontroller. OpenMSP430 has multiple clock modules and synchronizing modules which generate clock signals and take them. PT considers that

some of the synchronizing modules are gated by flip-flops and reports those paths as critical paths. To avoid this problem, we manually set them as false paths.

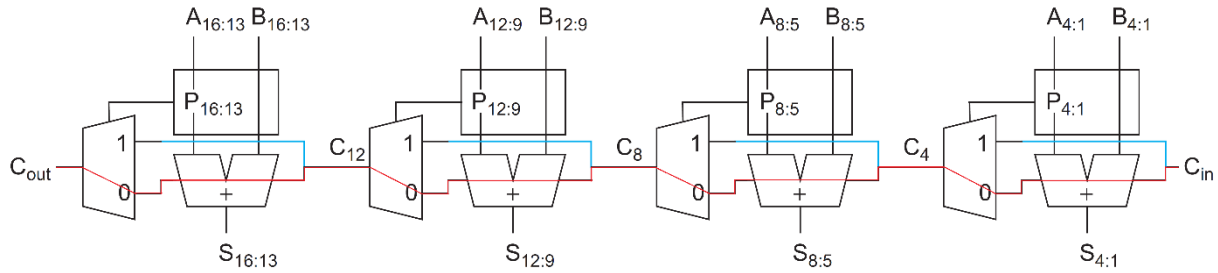


Figure 11. 16-bit carry skip adder *[8]*.

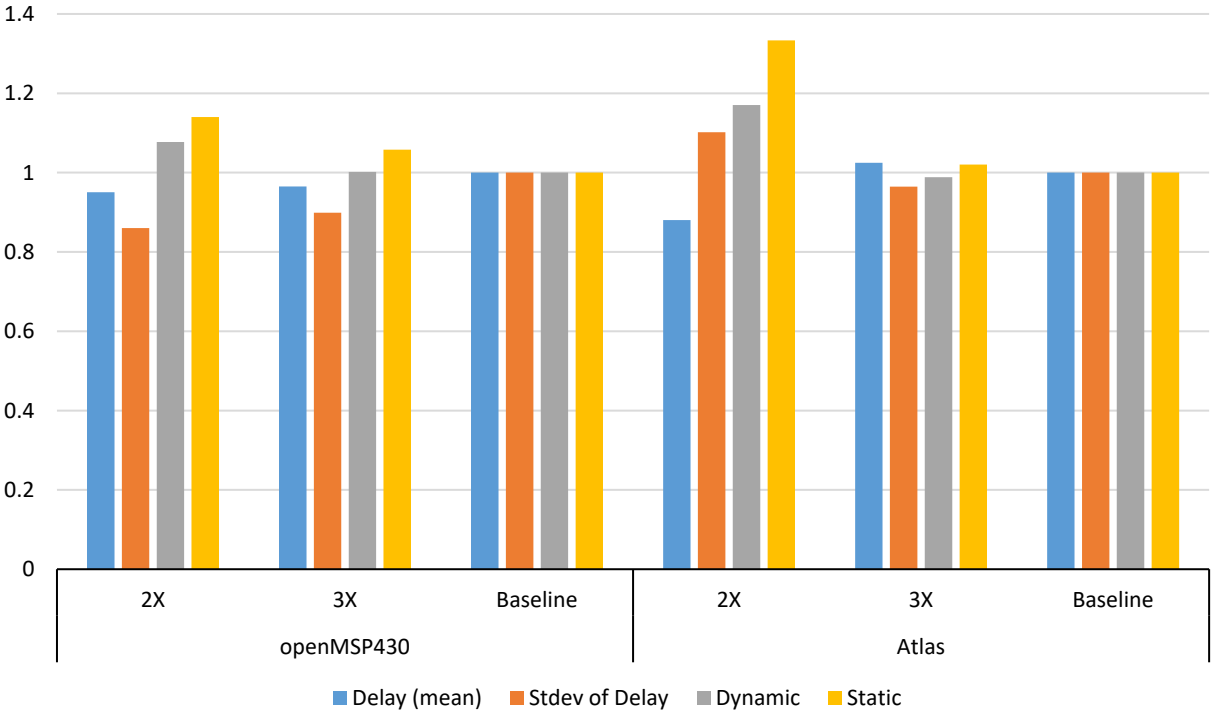


Figure 12. Synthesis results of the proposed designs and the baseline designs on both openMSP430 microcontroller and Atlas processor. 2X design is using only 2-input gates and 3X design is using both 2-input gates and 3-input gates. Baselines has access to all cells in the library. Results are normalized to baseline.

3.1.1.2 Evaluation

In Figure 12, we cannot see a clear tendency of constraining the library depending on the number of fan-in of cells. The s2X and 3X designs have lower delay in openMSP430 microcontroller, but the 3X design has higher delay in Atlas core. Furthermore, the 2X

design has lower delay in both cases, but it has significantly higher power consumption than baseline designs. One solution to this problem can be reducing the VDD of the 2X design further to reduce the power consumption while matching the delay to that of the baseline. However, the delay is greatly affected by the voltage reduction in the lower voltage region, and therefore VDD can be reduced only a little to maintain the delay lower than the baseline delay. In our testing environment, VDD can be reduced 4% at maximum. This can be calculated as ~8% dynamic power reduction from 0.5 V VDD, but it is still not enough for the 2X design to have a better power consumption than the baseline. Standard deviation of the delay path also fluctuates between different design points and show no clear tendency. Proposed designs have lower standard deviation value in openMSP430 microcontroller, but clearly not in Atlas core.

As a result, we conclude there is no strong correlation between the number of fan-in gates and the degree of random variation in delay. Removing large fan-in gates at the pre-synthesis stage has an unpredictable outcome which most likely depends on the input design.

3.1.2 Fine-Grained Gate Analysis

In the next step, we decide to analyze all of each cell's possible input to output transitions in terms of delay and the effect of random process variation on the delay. Instead of just naively removing large fan-in gates, we can use this information to tell us which gates are more susceptible to the random process variation in NTV and thus must be removed.

3.1.2.1 Experiment Setup

Before we start characterizing all possible transitions of logic gates, we first identify the types of transitions with the get_timing_paths command in PT. This command generally returns the slowest path only, but it can also return all possible paths with proper arguments. We use this command on all individual gates to find their possible transitions and directly feed them into the write_spice_deck command. Next, we sort the gates based

on their relative standard deviations of delays and make a sensitivity list. We first remove the top 10% of cells from the list and gradually increase the ratio. To reduce the time spent on simulation, we only simulate on openMSP430 microcontroller this time.

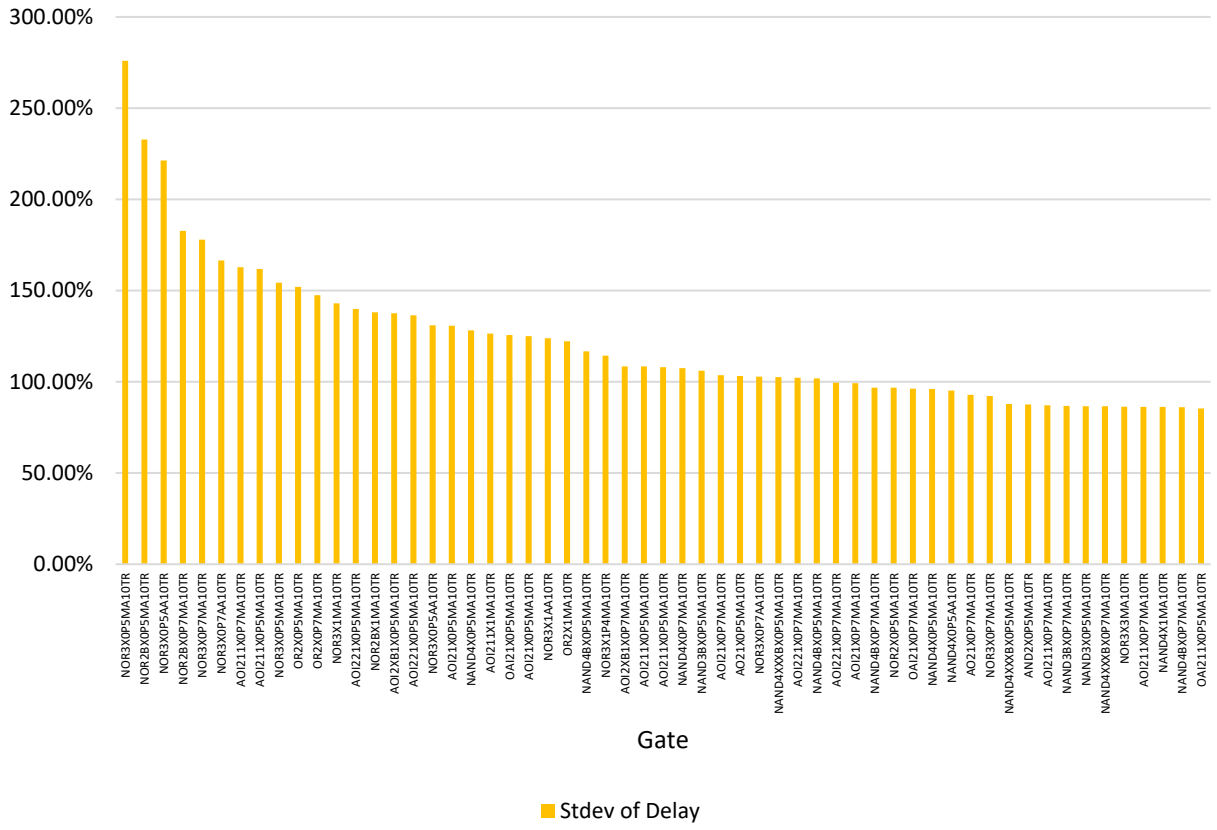


Figure 13. Snapshot of gate delay variation due to the random process variation at 0.5 V. Sorted based on relative standard deviation of delay.

3.1.2.2 Gate Characterization Result

From this experiment, we learned several interesting facts. First, there are gates which responds radically to random process variation. In Figure 13, one of the 3-input NOR gates has about 275% delay variation due to the random process variation. However, as we observe further, the degree of variation fades out very quickly. Around 20th position of the chart, the amplitude of the delay variation is decreased to 100%. After this point, the slope of decrement becomes almost horizontal, which means it is hard to expect to see a dramatic difference in delay variation from removing the most sensitive first 10%

and 20% of gates. Further, we need to consider the delay variation from dropping a supply voltage from 1.0 V to 0.5 V. Supply voltage of the given library is 1.0 V. While DC synthesizing, it optimizes design under this condition. However, our end goal is to run a synthesized circuit at 0.5 V. From this discrepancy, it is hard to synthesize a design which is optimized for low voltage operation. According to S. Jain et al. [2], multiplexers with wide fan-in or CMOS gates with long transistor chain are susceptible to the voltage change. This is observed in our gate characterization results in Figure 14.

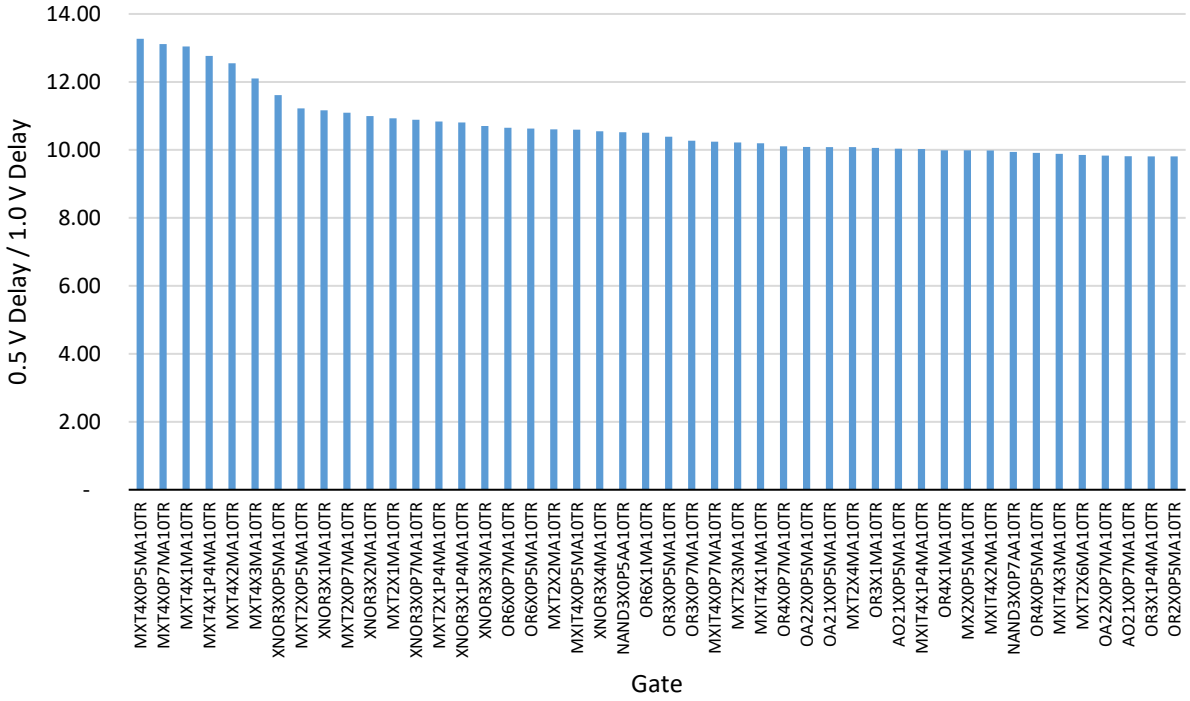


Figure 14. Snapshot of gate delay difference from supply voltage of 1.0 V to 0.5 V. Sorted from the largest to the smallest.

Unfortunately, not all the gate delays scale in the same manner. Some become much slower than the others while some are affected less. At this point, we do not have any method to separate these two types of gates. Library characterization can be a solution, but this requires help from TSMC to generate a new library which is targeted for 0.5 V.

3.1.2.3 Evaluation

Figure 15 shows the synthesis report of our new proposed designs. With removal of the most sensitive 10% of cells, delay has decreased a little but the standard deviation is increased about 10% more. In 20% and 30% designs, there are only demerits compared to the baseline. Especially in the 30% design, the dynamic power consumption has increased about 60% more compared to the baseline design. Removing some cells vulnerable to the random process variation not only increased the variation at the end, but also increased the power consumption greatly.

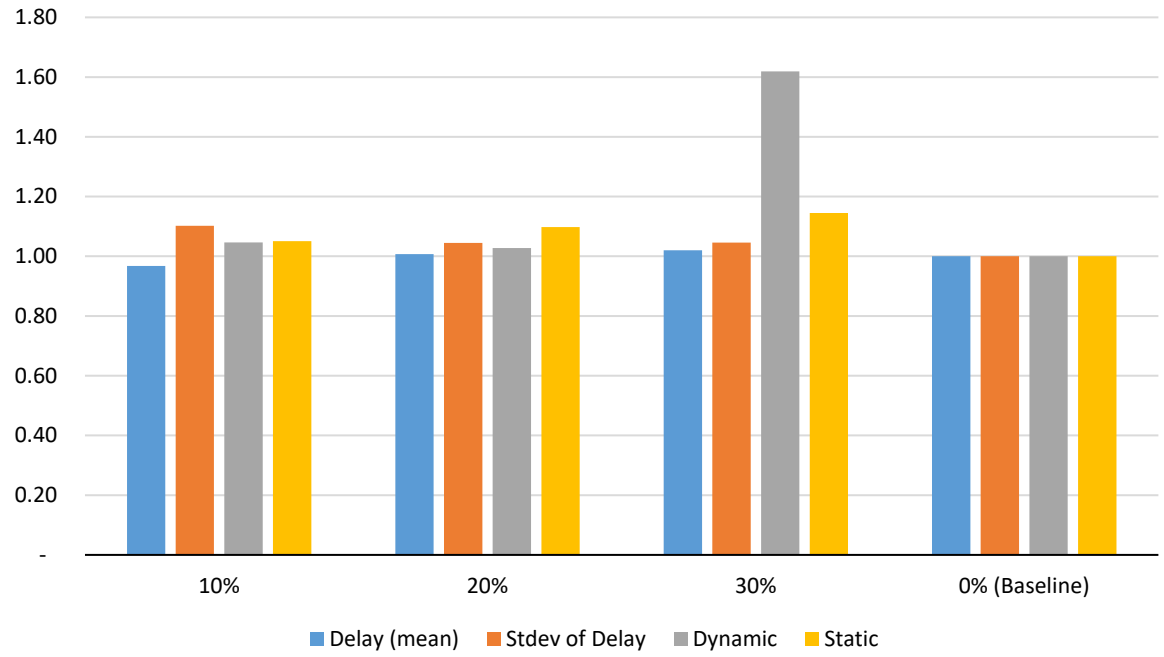


Figure 15. Synthesis results of proposed designs and baseline design. 10%, 20% and 30% denotes the portion of cells removed from the library. All results are normalized to baseline.

One possible reason for this result is that some cells are not replaceable. Our initial thought of this experiment was if we remove some unwanted cells from the library, the combination of other cells can replace the removed cell's role with very little penalty of power and delay. However, we can see that the removal of some cells ended up increasing power consumption radically. By observing differences between the 20%

design and the 30% design, we can assume some set of cells used to reduce the power consumption has been removed in the 30% design.

Another fact we overlooked in this experiment is that it is hard to identify if the critical path would have gone through the worst case transition of certain gates if we did not remove that gate from the library. To be more explicit, in CMOS 3-input NOR gate, input signals can be connected in any order without affecting the functionality. However, the transition speed from input A to output Y and input C to output Y can be different because the transistors in the pull up network (PUN) of NOR are connected in series. Likewise, the effect of random process variation on the gate delay also varies depending on which input port the critical path is connected to.

Table 2: Variation of 3-input NOR gate propagation delays on different transitions.

Inputs	Output Transition	
	Low -> High	High -> Low
A	30.73 %	55.96 %
B	31.51 %	130.94 %
C	33.38 %	221.34 %

Table 2 shows that the effect of random process variation is different for different transitions. We want the critical path to go through input A instead of B or C, but there is no certainty where DC will route the critical path. Under this circumstance, we cannot carelessly just remove this gate from the library before the synthesis. In the baseline design, if DC decides the critical path will go through input A, it will not be necessary to remove this gate from the library due to the low variation. Furthermore, as we have seen from Figure 15, this gate can be essential to optimize the power consumption from the perspective of DC.

Even if we assume that we can somehow precisely choose the "bad" cells before the synthesis, it is still questionable whether this method would be successful. When we analyze the critical path of the baseline design, we find only about 5% of the cells used in

the path are included in the top 40% of the sensitive cells. We conclude that this number is too small to make a meaningful outcome.

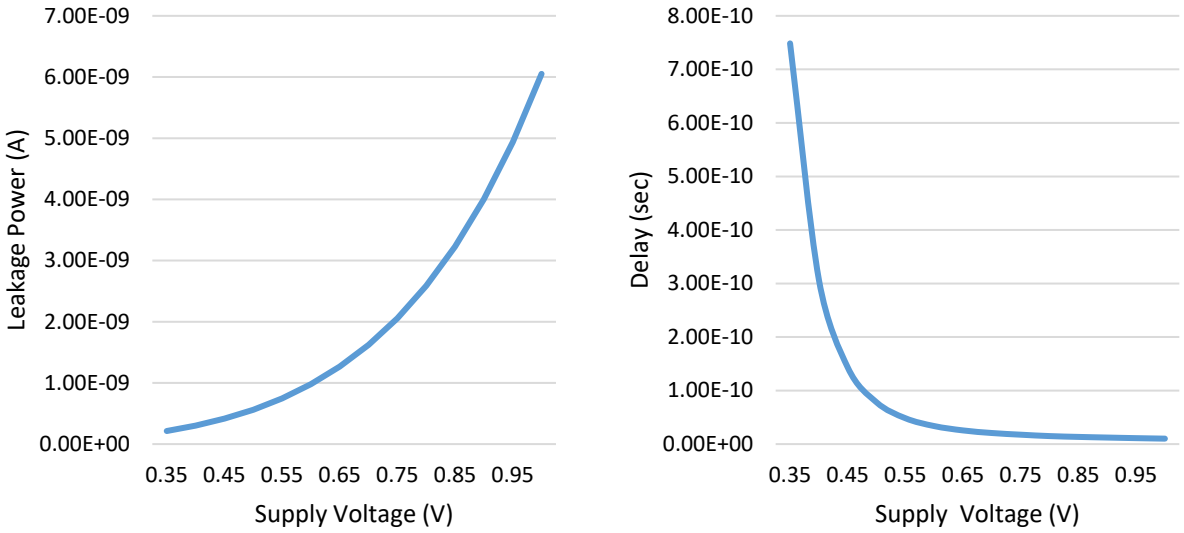
3.2 Timing Constraint Experiment

Our next experiment is to analyze the effect of synthesis timing constraint under NTV conditions. As we discussed in section 3.1.2.3, the library we have is characterized for 1.0 V and thus DC is not aware we are using the output design in lower input voltage. Because of the huge difference in the input voltage, we cannot guarantee the design optimized for 1.0 V also has a reasonable performance at a lower voltage. To address this issue, we first aim to synthesize a same design under a wide range of target frequency in a coarse-grained manner and measure the performance. Next, we do the same experiment with a narrower range of target frequency in a fine-grained manner.

3.2.1 Coarse-Grained Timing Constraint

In this experiment, we try to evaluate the relationship between different implementations and energy delay product (EDP). From previous experiments, we found that DC adaptively applies different implementations of given functions depending on the timing constraints. For example, at lower frequency, DC uses a ripple adder for the addition statement. If the target frequency is too high for the ripple adder to meet the timing constraint, DC starts to use more complicated adders such as carry lookahead adder or parallel prefix adder. By sweeping from low frequency to high frequency in a coarse-grained manner, we can see how using more complicated logics gives different EDP characteristics. A more complicated design should run faster than a simpler one, but it will consume more power due to its design complexity. However, if we lower the voltage of the complicated design to run at the same frequency as the simpler design, we might be able to lower the dynamic and static power significantly. With the result from this experiment, we can find if the more complicated design with shorter delay is more beneficial in a low frequency and low voltage environment. In general, the leakage

current is exponentially proportional to the input voltage, but delay also exponentially increases as the input voltage decreases [9]. Therefore, to process the same number of instructions, decreasing the input voltage will exponentially increase the execution time but at the same time the leakage current will exponentially decrease. Balancing between these two factors is the key to reduce the total power consumption while reducing the input voltage. For a quick case study, we characterize a single inverter over wide range of supply voltage. In Figure 16 (a), leakage power starts to sharply decrease as the supply voltage decreases from 1.0 V. In the contrast, the delay of the inverter sharply increases in the lower voltage region. If we multiply these two values to get a leakage energy consumed per transition, we get Figure 17. From Figure 17, we can assume there is an optimal supply voltage which minimizes the leakage energy. If the supply voltage is larger or smaller than this point, it is likely the circuit will start to consume more energy per cycle. Therefore, if the delay of the more complex design is equal to that of the simpler design outside this region, it will actually have worse leakage energy efficiency than it can achieve. Unfortunately, this is most likely to be the case. Assuming both designs have a similar leakage energy curve like Figure 17, the simpler design will have the best leakage energy efficiency around 0.6 V. To match with the delay of the simpler design, the more complex design can reduce its supply voltage further. At this point, the leakage energy of the complex design has higher leakage energy consumption per cycle. However, we still need to consider the dynamic energy. Unlike leakage energy, dynamic energy only decreases as the voltage decreases. If the dynamic energy is much higher than the leakage energy, lowering the voltage is still beneficial.



(a) (b)

Figure 16. Leakage power and delay characteristic of a single inverter over wide supply voltage range.

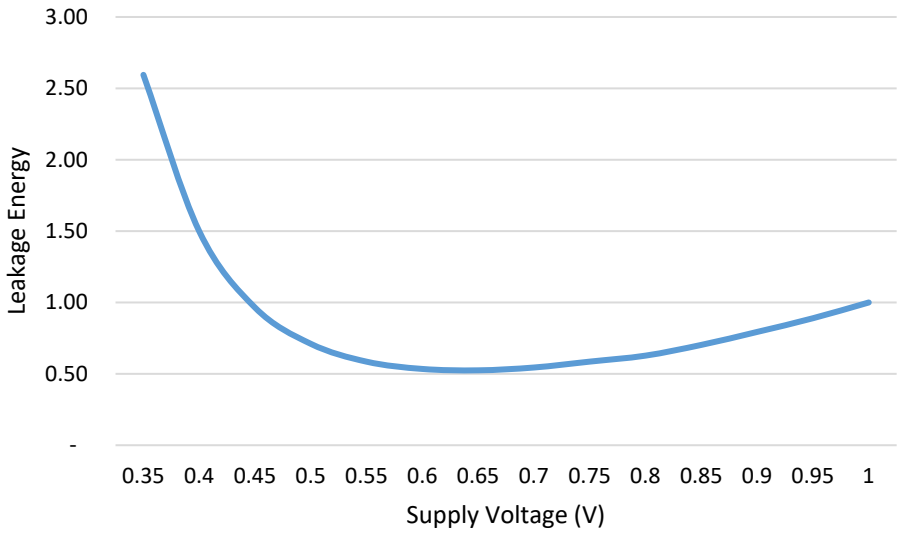


Figure 17. Leakage energy of single inverter. All data normalized to when supply voltage is 1.0 V.

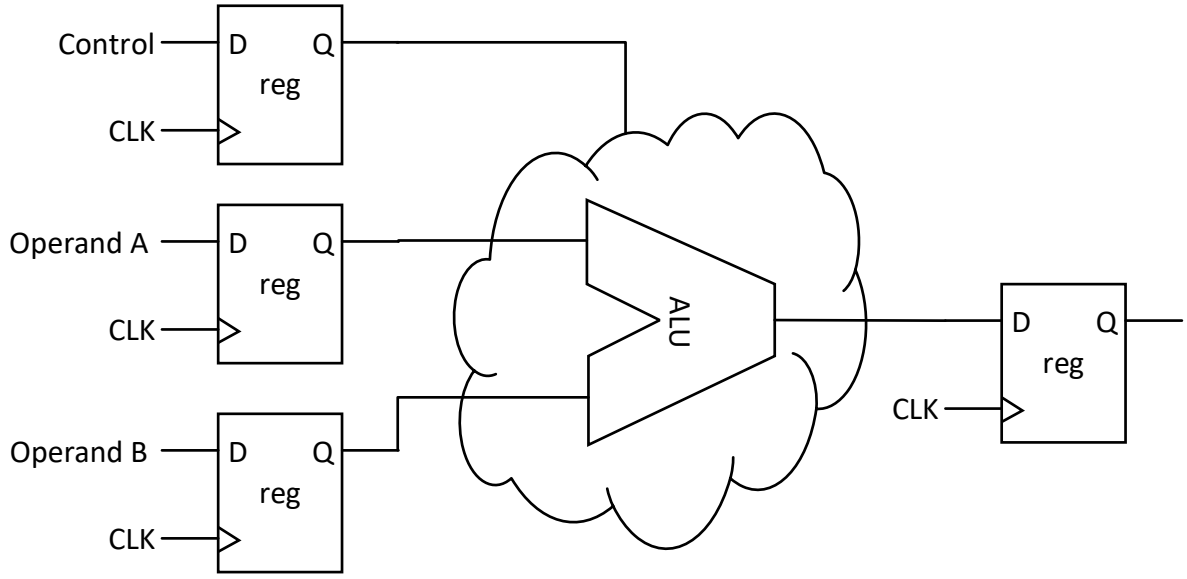


Figure 18. Test set-up for openMSP430 ALU.

3.2.1.1 Experiment Setup

From this experiment, we focus on the ALU part of the openMSP430 microcontroller. First, we found that all critical paths reported in previous experiments are going through the ALU, which is taking about 80% of the total length of the critical path. And second, more importantly, the maximum frequency of openMSP430 is severely limited by the memory access time. OpenMSP430 has no cache and no stalling mechanism for the memory access. Instead of stalling, openMSP430 requires operating frequency to be low enough to make a memory access within one cycle. Because of this limitation, openMSP430 can be synthesized only up to 250 MHz. From 0 to 250 MHz of frequency range, it is hard to see a noticeable difference in implementation. Therefore, we pull out the ALU part which can be synthesized up to 1.1 GHz independently. In the original design, ALU itself is not gated by registers so we add input and output registers (Figure 18), easing measurement of the critical path in the Synopsys tools. Once syntheses are done, we gradually lower the input voltage of each design from 1.0 V to 0.4 V and measure leakage current and delay. To capture the effect of random process variation also, we use a new metric to evaluate the delay. Instead of directly using the mean delay

value from the Monte Carlo simulation, we calculate the delay value at 3-sigma point. This will give us a rough idea at what frequency the given design can safely run even with the variation. After gathering this information, we can find the minimum voltage required to run each design at a certain frequency. Dynamic power is calculated by using eq. 1. We sweep frequency from 250 MHz to 1 GHz with a 250 MHz tick.

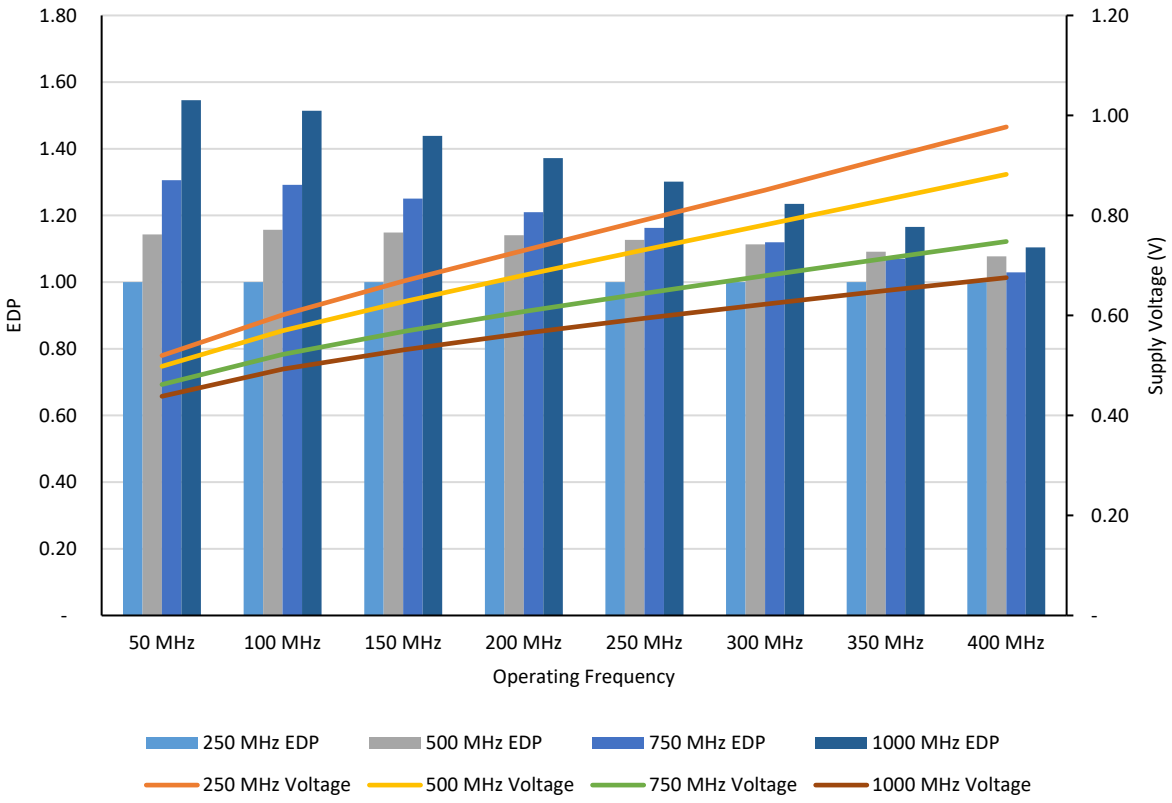


Figure 19. EDPs and required supply voltages of four different designs at ISO-performance points. X-axis denotes the operating frequency. Lines and bars denote different designs synthesized for different frequencies at 1.0 V. EDP data normalized to 250 MHz design.

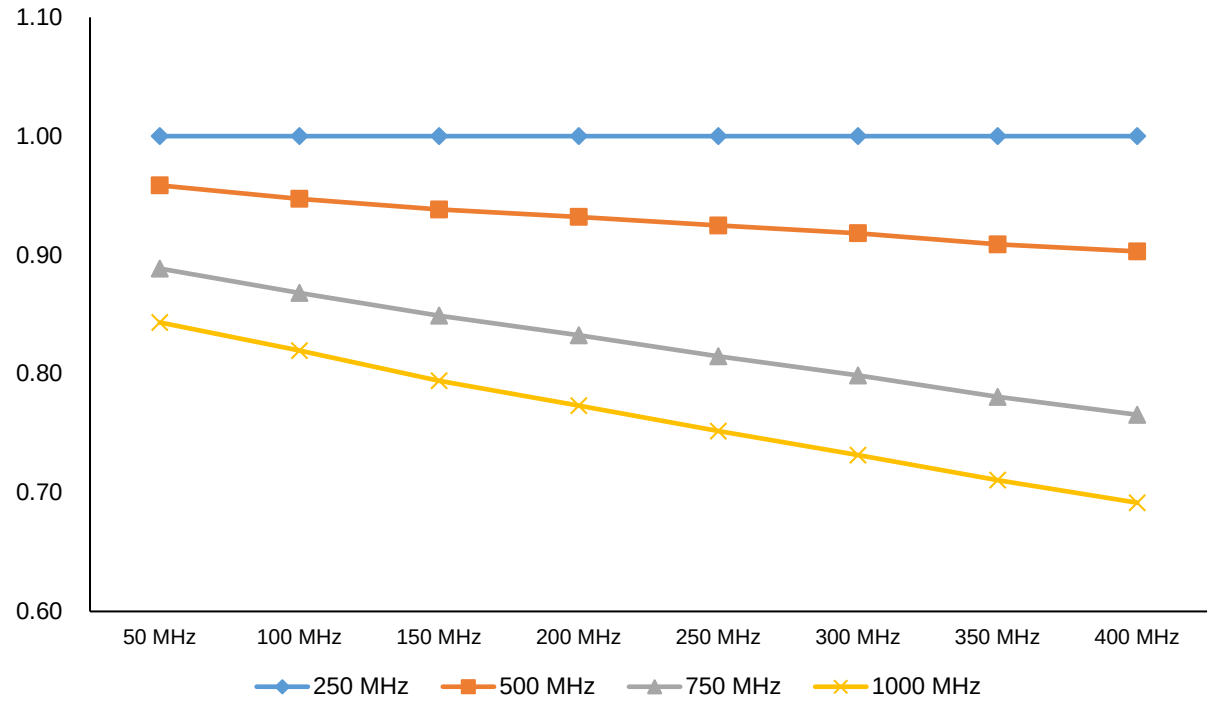


Figure 20. Supply voltage value of each design under different operating frequencies. Data normalized to 250 MHz design.

3.2.1.2 Evaluation

After measuring delays, interestingly we found that the 250 MHz design can run up to 400 MHz even if it is synthesized for lower frequency. In Figure 19, the efficiency of the 250 MHz design drops sharply compared to the others as the operating frequency increases. However, at lower frequency the 250 MHz design dominates other designs. The voltage difference between the 250 MHz design and other designs is huge at higher frequency, but becomes dramatically smaller at lower frequency. This tendency is clearly shown in Figure 20. The 1000 MHz design needs only 70% of the supply voltage of the 250 MHz design to work on 400 MHz, but it requires 84% of supply voltage of 250 MHz design to work at 50 MHz. Considering that we plan to use these designs in a low voltage and low frequency environment, it is always beneficial to use the minimal design to achieve better EDP.

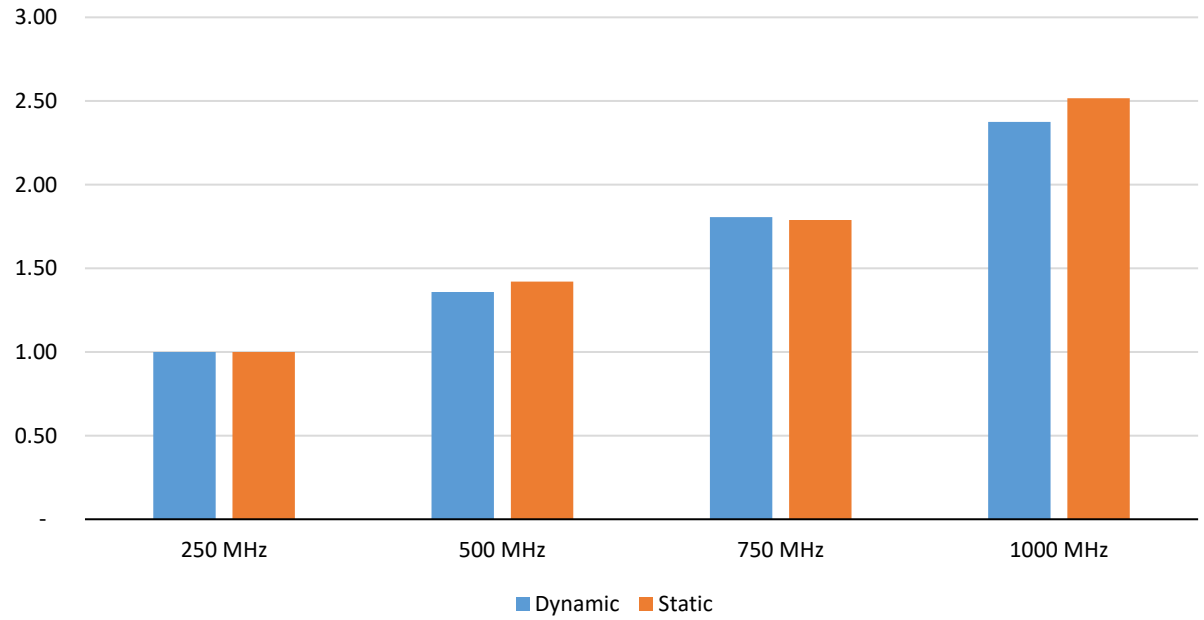


Figure 21. Dynamic power and static power of 250, 500, 750, and 1000 MHz designs all running at the same frequency. Voltage scaling is not applied here. All data normalized to 250 MHz design.

At the same operating frequency, the more complicated design requires much lower supply voltage than the simpler design, but the penalty from the design area eclipses the benefit from the low supply voltage. When all designs are running at 1.0 V supply voltage, the power consumption increases about by 40%, 75%, and 150% from 250 MHz design to the 500 MHz, 750 MHz, and 1000 MHz designs, respectively, in Figure 21.

3.2.2 Fine-Grained Timing Constraint

In this experiment, we aim to find how changing the sizes of cells of a given design without changing the structure can affect EDP. While we increase or decrease, the target frequency very slightly, DC swaps some gates with bigger or smaller gates without changing the structure at all. Bigger gates are less vulnerable to the process variation than the smaller gates, but they consume more power. Furthermore, with insufficient driving force, they are slower than the smaller gates. From this experiment, we observe the effect of gate sizing more systematically.

3.2.2.1 Experiment Setup

Overall setup is equal to section 3.2.1.1, but here we decrease the delta from 250 MHz to 20 MHz and sweep from 250 MHz to 350 MHz. We do not target frequency lower than 250 MHz since in DC there is no design change below this point.

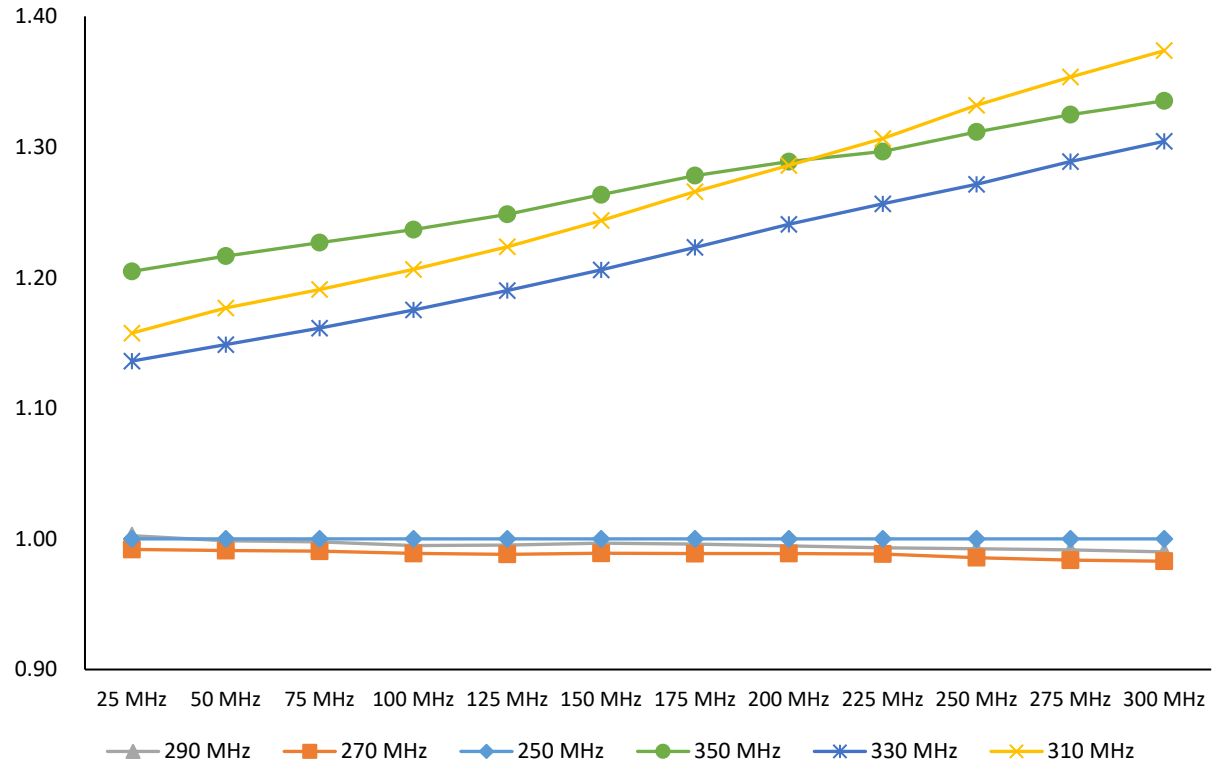


Figure 22: EDPs of six designs at ISO-performance points. X-axis denotes the operating frequency. Each line denotes different designs synthesized for different frequencies at 1.0 V. Data normalized to 250 MHz design.

3.2.2.2 Evaluation

In Figure 22, we can group designs into two implementations. The 250 – 290 MHz designs share the same structure, and the 310 – 350 MHz designs share another structure. Within the 250 – 290 MHz group, it is hard to see clear differences. The 270 MHz design is generally better than the 250 MHz design, but the difference is only about 1 – 2%. This difference is even smaller at lower frequency. EDP graphs of 310 – 350 MHz designs are rather confusing.

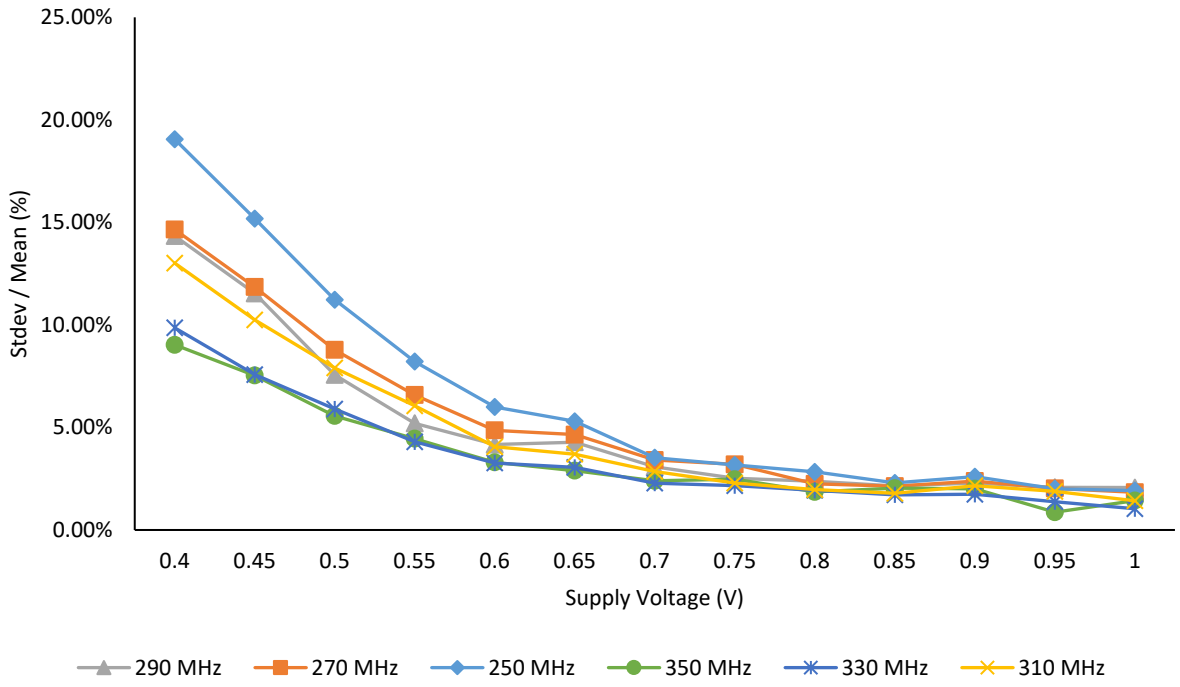


Figure 23: Relative standard deviation of delay over different supply voltages. Each line denotes different designs synthesized for different frequencies.

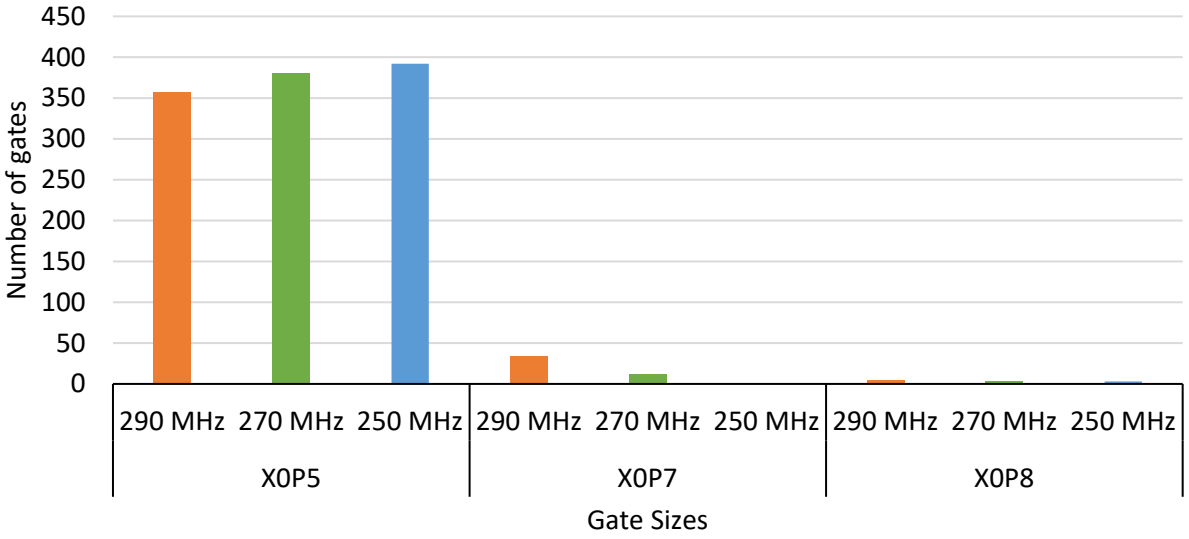


Figure 24. Distribution of gate sizes over 250 MHz, 270 MHz, and 290 MHz designs. X0P5 is a small size, X0P7 is a middle size, and X0P8 is a large size.

The 250 – 290 MHz designs have smaller EDP than the 310 – 350 MHz designs, but they are more susceptible to random process generation in the NTV environment. In Figure 23, the critical path of the 250 MHz design varies about twice as much at lower voltage

than the 350 MHz design. Figure 24 shows that what we expected is correct. As we increase the frequency slowly, the synthesized design replaces the smaller gates with larger gates. The distribution of this replacement rather depends on DC, but we can see that this is certainly affecting the critical path. This is understandable since the critical path is the path which DC tries to optimize the most, and thus DC will only replace the gates on the critical path. However, the base power consumptions of the 250 – 290 MHz designs are much lower, and this gives us lower EDP. The degree of variation is too small to have a meaningful effect on the designs. If we assume a gate has mean delay value of μ and standard deviation of σ, the relative standard deviation is $\frac{\sigma}{\mu}$. If we assume a number n of these gates are connected in series, the new mean is $n\mu$ and the new standard deviation is $\sqrt{n}\sigma$. Therefore, we can say the relative standard deviation is proportional to $\frac{1}{\sqrt{n}}$. In the current design, the length of the critical path is on average 40 gates, which is long enough to suppress the effect of random process variation. Increasing the number of pipeline stages and making each stage's depth as shallow as 5 to 10 gates can increase the effect of the random process variation, but this method will also give us a design with more pipeline registers and more power consumption. The portion of leakage energy from total energy consumption in NTV is not negligible and therefore increasing the size of the circuit is strongly discouraged.

3.3 Exploiting Process Variation

The idea of process variation exploitation relies on delay variation between different pipeline stages. When each stage is supposed to have the same delay at synthesis point, this can change after manufacturing.

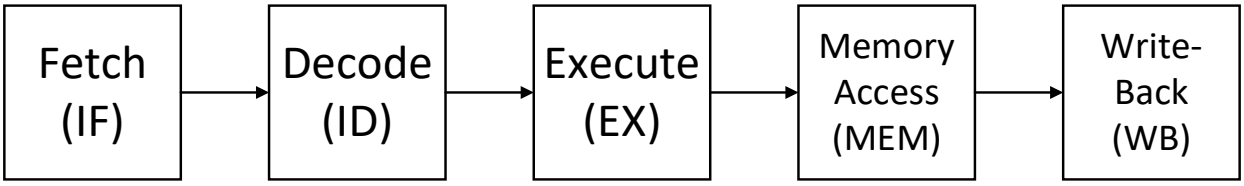


Figure 25. Pipeline of 5-stage in-order processor.

Say there is a simple 5-stage pipeline like Figure 25. If we assume there is no process variation and thus no delay variation, the timing balance between different stages would look like Figure 26. However, because of the process variation, some stages will have longer delay, and some shorter, than expected (Figure 27).

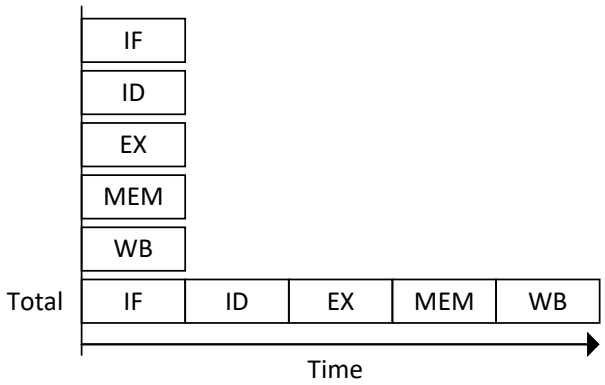


Figure 26. Ideal timing balance between different stages.

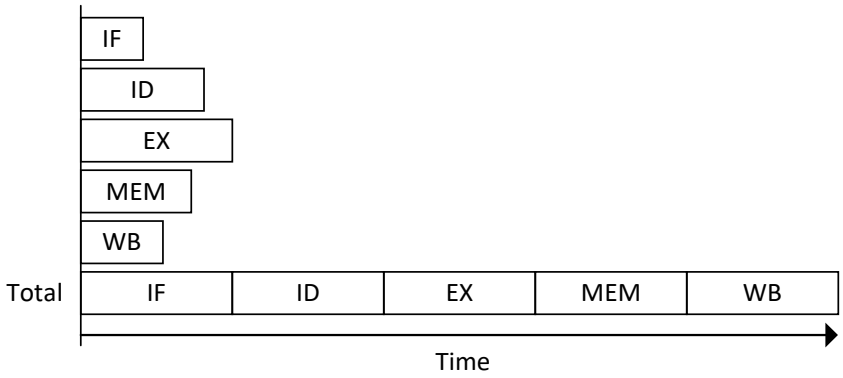


Figure 27. Timing balance between different stages with process variation.

All stages should have the same timing constraint, so the timing constraints of all stages are forced to be fixed to the longest. In this situation, providing the same VDD to all stages can be wasteful. To solve this problem, we considered installing LDO per stage. In this case, we can control each stage's VDD and provide only the minimum required voltage. However, this method is problematic since it is hard to define the border of a stage. Figure 25 oversimplifies a pipelined processor. In a real one, multiple signals connect different stages. For example, the control module of the Atlas processor sends and receives signals from all stages. Also, there is a forwarding unit and a branch prediction unit that require

inter-stage communication. Because of these signals, the definition of critical path per stage becomes very vague. Changing the supply voltage of a certain stage directly affects the path delay of other stages. The Atlas processor is a fairly simple core with which we can solve this problem by iteratively changing the supply voltages of different stages. However, as the design becomes more and more complicated with more interconnections between different stages, this approach will encounter significant problems. Therefore, we withdraw the idea of setting LDO per stage.

4. Conclusion

We confirmed that the process variation in NTV is more severe than in the super-threshold voltage environment, but it is still not large enough to make a meaningful difference in the real design. The delay variation due to the random process variation greatly depends on the logic depth. The path which consists of only a few gates will suffer greatly from the process variation, but having only few gates also means it is very unlikely to be a critical path. In our experiments, we have seen at most 13% delay variation due to the process variation. In NTV, it is trivial to reduce delay by 13% by increasing VDD slightly. As we have seen in chapter 1, the delay value changes greatly with small delta of VDD in NTV, and therefore power consumption also does not increase considerably.

We also found that synthesizing designs with minimal timing constraints gives us the best energy efficiency at NTV. These minimal designs are more susceptible to random process variation than more complex designs, but their energy efficiencies are superior to the rest. Even after increasing voltage to suppress the effect of random process variation, the minimal designs gave us the best energy efficiencies.

References

[1] R. G. Dreslinski, M. Wieckowski, D. Blaauw, D. Sylvester and T. Mudge, "Near-threshold computing: Reclaiming Moore's law through energy efficient integrated circuits," in *IEEE 98.2*, 2010.

[2] S. Jain et al., "A 280mV-to-1.2V wide-operating-range IA-32 processor in 32nm CMOS," *ISSCC Dig. Tech,* pp. 66-68, 2012.

[3] T. Abhishek, S. R. Sarangi and J. Torrellas, "ReCycle: Pipeline adaptation to tolerate process variation," *ACM SIGARCH Computer Architecture News,* vol. 35, no. 2, 2007.

[4] G. Yip, "Expanding the synopsys primetime solution with power analysis," Synopsys, 2006.

[5] C. W. Gear, "Simultaneous numerical solution of differential-algebraic equations," *IEEE Transactions on Circuit Theory*, vol. 18.1, pp. 89-95, 1971.

[6] O. Girard, "Opencores," [Online]. Available: http://opencores.org/project,openmsp430, 2009.

[7] S. Nolting, "Opencores," [Online]. Available: http://opencores.org/project,atlas_core, 2013.

[8] N. Weste and D. M. Harris, *CMOS VLSI Design*, Massachusetts: Pearson, 2011.

[9] G. Eneman et al., "Si1-xGex-channel PFETs: Scalability, layout considerations and compatibility with other stress techniques," *ECS Transactions,* vol. 35.3, pp. 493-503, 2011.

Appendix – Tcl Scripts

```
source "library.tcl"

set DESIGN_NAME "wrapper"

set iter 1
set path_num 10
set link_path "* $LIB_WC_FILE"

set subckt $LIB_PATH/tsmc65_${VTH}_sc_adv10_mismatch.lpeSpc
set header $LIB_PATH /tsmc65_lib.spi

for {set i 0} {$i < $iter} {incr i} {

        set trgt_dir ./results/$DESIGN_NAME\_$gate_constraint\_$i
        read_verilog $trgt_dir/$DESIGN_NAME.gate.v

        link_design $DESIGN_NAME

        set supply_voltage 1.0

        read_sdc $trgt_dir/${DESIGN_NAME}.sdc
        read_sdf $trgt_dir/${DESIGN_NAME}.sdf

        source constraints_0p5v.tcl
        set j 0

        foreach_in_collection paths [get_timing_paths -delay_type max -max_paths 9
-nworst 3 -slack_greater_than 0] {

                set sub_dir $trgt_dir/path_$j
                file mkdir $sub_dir

                incr j

                report_timing -input_pins -significant_digit 4 >
$sub_dir/timing_0p5v.rep

                set start [get_object_name [get_attribute $paths startpoint]]
                set end [get_object_name [get_attribute $paths endpoint]]

                set delay_list "{delay $start $end}"

                write_spice_deck -o $sub_dir/0p5v -header $header
                                    -sub_circuit_file $subckt \
                                 -logic_one_name vdd \
                                    -logic_one_voltage $supply_voltage \
                                 -logic_zero_name vss \
                                 -logic_zero_voltage 0 \
                                 -transient_step 0.0005 \
                                 $paths \
                                 -user_measures $delay_list
        }

        remove_design -hierarchy
}

quit
```

Figure 28. 'write_spice_deck' command used in PT Tcl script.

```
set DESIGN_ITER 1
set PATH_ITER 1
set WITH_DC_ULTRA 1

set_host_options -max_cores 4

source -echo -verbose ./library.tcl
source -echo -verbose ./read.tcl
source -echo -verbose ./constraints.tcl

set_wire_load_model -name $LIB_WIRE_LOAD
set_wire_load_mode top


for {set i 0} {$i < $DESIGN_ITER} {incr i} {

        #No 4X
        if { $gate_constraint == "2X" } {
                set_target_library_subset -dont_use {"*3X*" "*4X*" "AO*" "OA*"
"*3BX*" "*4BX*" "*ADDF*" "MX*"}
        } elseif { $gate_constraint == "3X" } {
                set_target_library_subset -dont_use {"NOR3X*" "OR6X*" "MXT2X*"
"MXIT2X*" "AOI221X*" "AOI222X*" "OR3X*" "AOI211X*"}
        }

        dont_touch_network {clk}

        # Prevent assignment statements in the Verilog netlist.
        set_fix_multiple_port_nets -all -buffer_constants

        # Configuration
        current_design $DESIGN_NAME
        set_max_area  0.0
        set_flatten false
        set_structure true -timing true -boolean false

        # Synthesis
        if {$WITH_DC_ULTRA} {
                compile_ultra       -area_high_effort_script -no_autoungroup -
no_boundary_optimization
        } else {
                compile             -map_effort high -area_effort high
        }

        set TRGT_DIR ./results/${DESIGN_NAME}_${gate_constraint}_$i

        current_design $DESIGN_NAME

        define_name_rules verilog -case_insensitive
        change_name -rules verilog -hierarchy

        write_script -hier -o $TRGT_DIR/$DESIGN_NAME.tcl

        write_sdf $TRGT_DIR/$DESIGN_NAME.sdf
        write_sdc $TRGT_DIR/$DESIGN_NAME.sdc
        write -hierarchy -format verilog -o "$TRGT_DIR/$DESIGN_NAME.gate.v"
        write -hierarchy -format ddc -o "$TRGT_DIR/$DESIGN_NAME.ddc"
}

quit
```

Figure 29. Large fan-in cell constraint experiment Tcl script.

```
import sys
import os
import re

if len(sys.argv) < 3:
        sys.exit('Not enough minerals')

cell_list = open(sys.argv[2], 'r')
timing_list = open('variation.txt', 'w')

for cell in cell_list:
        os.system('rm test_bd.v')
        verilog = open(sys.argv[1], 'r')
        output = open('test_bd.v', 'w')

        for module in verilog:
                if cell.strip() in module:
                        m = re.search('\(.*\)', module)
                        pin = ((m.group(0)[1:-1]).replace(' ', '')).split(',')
                        break
        temp = []
        for port in pin:
                temp.append('.' + port + '(' + port + ')')

        output.write('module test_bd (' + ', '.join(pin) + ');\n')
        output.write('\tinput ' + ', '.join(pin[1:]) + ';\n')
        output.write('\toutput ' + pin[0] + ';\n')
        output.write('\t' + cell.strip() + ' INST ' + '(' + ', '.join(temp) +
');\n')
        output.write('endmodule')
        output.flush()
        os.system('pt_shell -f library.tcl')
        count = open('spice_count', 'r')
        sim_count = int(count.readline())

        for i in range(sim_count):
                os.system('cat add.txt >> 0p5v_' + str(i) + '_stim')
                os.system('hspice -mt 8 -i 0p5v_' + str(i) + ' -o 0p5v_' + str(i))

        f = open('timing', 'r')

        input_port = []
        input_trans = []
        output_port = []
        output_trans = []

        for i in range(sim_count):
                for line in f:
                        if '(in)' in line:
                                elem = line.split()
                                input_port.append(elem[0])
                                input_trans.append(elem[-1])
                        elif '(out)' in line:
                                elem = line.split()
                                output_port.append(elem[0])
                                output_trans.append(elem[-1])
                                break

        for i in range(sim_count):

                f = open('0p5v_' + str(i) + '_mpp0', 'r')
```

Figure 30. Per-cell characterization Python script.

```
set supply_voltage 0.5
read_verilog test_bd.v

link

set_max_delay 4 -from [all_inputs] -to [all_outputs]

set j 0

foreach_in_collection paths [get_timing_paths -delay_type max -nworst 20 -
slack_greater_than 0] {

        set start [get_object_name [get_attribute $paths startpoint]]
        set end [get_object_name [get_attribute $paths endpoint]]
        set delay_list "{delay $start $end}"
        echo $delay_list

        get_attribute $paths arrival >> pt_timing

        write_spice_deck -o 0p5v_$j -header $header -sub_circuit_file $subckt \
                         -logic_one_name vdd -logic_one_voltage $supply_voltage \
                                         -transient_step 0.0001 \
                         -logic_zero_name vss -logic_zero_voltage 0 $paths \
                         -user_measures $delay_list
        incr j
}

echo $j > spice_count

report_timing -nworst 20 -slack_greater_than 0.0 > timing

quit
```

Figure 31. Per-cell characterization Tcl script.

```
#!/bin/bash

TRGT_DIR=./results/wrapper_4X_0
rm timing.txt
rm power.txt
rm leak.txt
rm -r results/*/

for i in 4.0 2.0 1.3333 1.0
do
        echo "$i" >> timing.txt
        echo "$i" >> leak.txt
        echo "$i" >> power.txt

        sed -i "25 s/\(CLOCK_PERIOD \).*/\1$i/" constraints.tcl
        dc_shell-xg-t -f stat_synthesis.tcl

        sed '/module wrapper/,/endmodule/d' $TRGT_DIR/wrapper.gate.v >
$OPENMSP430/openmsp430/trunk/core/rtl/verilog/omsp_alu.v

        (cd $OPENMSP430/openmsp430/trunk/core/sim/rtl_sim/run/; ./run_c coremark_v1.0)

        pt_shell -f pts.tcl

        nettran -cdl $TSMC65GP_LIB/tsmc65_rvt_sc_adv10_mc.lpeSpc -verilog
$OPENMSP430/openmsp430/trunk/core/rtl/verilog/omsp_alu.v -verilog-b0 vss -verilog-b1 vdd -
outType spice -outName ./results/omsp_alu

        cp ./results/test_bd.spi $TRGT_DIR
        sed -n '/\.SUBCKT omsp_alu/,/\.ENDS/p' ./results/omsp_alu > $TRGT_DIR/omsp_alu.spi

        for j in 0.4 0.45 0.5 0.55 0.6 0.65 0.7 0.75 0.8 0.85 0.9 0.95 1.0
        do
                sed -i "18 s/\(supply_voltage \).*/\1$j/" stat_pts.tcl

                # Default clock period for spice simulation is 50ns
                sed -i "25 s/\(CLOCK_PERIOD \).*/\150/" constraints_0p5v.tcl
                pt_shell -f stat_pts.tcl
                gawk -i inplace '/^+ trig/ { var = $9 }; {gsub(/td = .*ns/, "td = " var);
print}' ./results/wrapper_4X_0/path_*/*v_stim

                for k in {0..8}
                do
                        echo ".option method=gear" >> $TRGT_DIR/path_$k/0p5v_stim
                        hspice -mt 4 -i $TRGT_DIR/path_$k/0p5v -o $TRGT_DIR/path_$k/0p5v
                done

                mkdir $TRGT_DIR/$j
                gawk '/25.000/ {print $1}' $TRGT_DIR/path*/0p5v.mt0 > $TRGT_DIR/$j/delay.txt
                var1=$(sort -gr $TRGT_DIR/$j/delay.txt | head -1)
                echo $var1 >> timing.txt

                # Re-run primetime to generate a spice deck with customized simulation time
                var2=$(python convert.py $var1)
                sed -i "25 s/\(CLOCK_PERIOD \).*/\1$var2/" constraints_0p5v.tcl
                pt_shell -f stat_pts.tcl
                sed -i -e '/\.vdd/d' $TRGT_DIR/path*/0p5v
                gawk -i inplace '/^+ trig/ { var = $9 }; {gsub(/td = .*ns/, "td = " var);
print}' ./results/wrapper_4X_0/path_*/*v_stim

                mv $TRGT_DIR/path* $TRGT_DIR/$j
        done

        gawk 'FNR==26,FNR==28 {print $5}' $TRGT_DIR/pt_power.rep >> power.txt

        mkdir ./results/$i
        mv $TRGT_DIR ./results/$i
done
```

Figure 32. Timing constraint change experiment shell script.

```
source "library.tcl"

set DESIGN_NAME omsp_alu

read_verilog $OPENMSP430/openmsp430/trunk/core/rtl/verilog/$DESIGN_NAME.v

set link_path "* $LIB_WC_FILE"

link_design $DESIGN_NAME

report_timing -input_pins -significant_digit 4 > timing.rep

set_wire_load_model -name $LIB_WIRE_LOAD

set power_enable_analysis true
set power_analysis_mode time_based

read_vcd $OPENMSP430/openmsp430/trunk/core/sim/rtl_sim/run/tb_openMSP430.vcd -strip_path
tb_openMSP430/dut/execution_unit_0/alu_0 -time {2292000 16629200}
update_power
report_power > ./results/wrapper_4X_0/pt_power.rep

quit
```

Figure 33. Timing constraint change experiment power report Tcl script.

```
import os
import sys

temp=sys.argv[1]
print float(temp)*1000000000*1.2
```

Figure 34. Timing constraint change experiment Python script.

```
#!/bin/bash

VOLT_LIST="0.4 0.45 0.5 0.55 0.6 0.65 0.7 0.75 0.8 0.85 0.9 0.95 1.0"
SPEED_LIST="4.0 2.0 1.3333 1.0"

rm delay_list.txt

for SPEED in $SPEED_LIST
do
    echo $SPEED >> delay_list.txt

    for VOLT in $VOLT_LIST
    do
        WORK_DIR="./$SPEED/wrapper_4X_0/$VOLT"
        rm $WORK_DIR/var_delay.txt

        for i in {0..2}
        do
            CURR_DIR="$WORK_DIR/path_$i"
            sed -i 's/\(\.tran.*ns.*ns\).*/\1 sweep monte=200/' $CURR_DIR/0p5v_stim
            sed -i '/^\.option method=gear/,$ d' $CURR_DIR/0p5v_stim
            cat add.txt >> $CURR_DIR/0p5v_stim

            hspice -dp 40 -dpconfig sge.cfg -i $CURR_DIR/0p5v -o $CURR_DIR/0p5v
            gawk '/User_Specified_Quantiles/,/end_of_User_Specified_Quantiles/ {if
(index($0,"delay_") > 0) {print $3}}' $CURR_DIR/0p5v.mpp0 >> $WORK_DIR/var_delay.txt
        done

        sort -gr $WORK_DIR/var_delay.txt | head -1 >> delay_list.txt

    done
done
```

Figure 35. Monte Carlo simulation shell script.

```
#!/bin/bash

SPEED_LIST="4.0 2.0 1.3333 1.0"

rm leak_map.txt

for SPEED in $SPEED_LIST
do

    echo $SPEED >> leak_map.txt
    for i in {1..10}
    do

        var=$(sed -n "${i}p" rand_vec.vec)
        sed -i "8 s/.*/$var/" $SPEED/omsp_alu.vec
        hspice -i $SPEED/wrapper_4X_0/test_bd.spi -o $SPEED/wrapper_4X_0/test_bd
        sed -n '/400.00000m/,/1.00000/p' $SPEED/wrapper_4X_0/test_bd.lis >>
leak_map.txt

    done
done
```

Figure 36. Leakage power experiment shell script.

```
.lib "$TSMC65GP_MODEL/CMN65GPLUS_2d5_lk_v1d4_usage.l" tt_lib

.include "$TSMC65GP_LIB/tsmc65_lvt_sc_adv10_mismatch.lpeSpc"
.include "$TSMC65GP_LIB/tsmc65_rvt_sc_adv10_mismatch.lpeSpc"

.vec "../omsp_alu.vec"
.include "./omsp_alu.spi"

.param input = 1.0

.GLOBAL vdd vss

vvdd vdd 0 input
vvss vss 0 0

XU99999 alu_out[15] alu_out[14] alu_out[13] alu_out[12] alu_out[11] alu_out[10]
alu_out[9] alu_out[8] alu_out[7] alu_out[6] alu_out[5] alu_out[4] alu_out[3]
alu_out[2] alu_out[1] alu_out[0] alu_out_add[15] alu_out_add[14] alu_out_add[13]
alu_out_add[12] alu_out_add[11] alu_out_add[10] alu_out_add[9] alu_out_add[8]
alu_out_add[7] alu_out_add[6] alu_out_add[5] alu_out_add[4] alu_out_add[3]
alu_out_add[2] alu_out_add[1] alu_out_add[0] alu_stat[3] alu_stat[2] alu_stat[1]
alu_stat[0] alu_stat_wr[3] alu_stat_wr[2] alu_stat_wr[1] alu_stat_wr[0] dbg_halt_st
exec_cycle inst_alu[11] inst_alu[10] inst_alu[9] inst_alu[8] inst_alu[7]
inst_alu[6] inst_alu[5] inst_alu[4] inst_alu[3] inst_alu[2] inst_alu[1] inst_alu[0]
inst_bw inst_jmp[7] inst_jmp[6] inst_jmp[5] inst_jmp[4] inst_jmp[3] inst_jmp[2]
inst_jmp[1] inst_jmp[0] inst_so[7] inst_so[6] inst_so[5] inst_so[4] inst_so[3]
inst_so[2] inst_so[1] inst_so[0] op_dst[15] op_dst[14] op_dst[13] op_dst[12]
op_dst[11] op_dst[10] op_dst[9] op_dst[8] op_dst[7] op_dst[6] op_dst[5] op_dst[4]
op_dst[3] op_dst[2] op_dst[1] op_dst[0] op_src[15] op_src[14] op_src[13] op_src[12]
op_src[11] op_src[10] op_src[9] op_src[8] op_src[7] op_src[6] op_src[5] op_src[4]
op_src[3] op_src[2] op_src[1] op_src[0] status[3] status[2] status[1] status[0]
omsp_alu

.option method=gear
.dc input 0.4 1.0 0.05
.print i(vvdd)
```

Figure 37. Leakage power experiment testbench.