Mitigating NBTI in the Physical Register File through Stress Prediction

Saurabh Kothawade*, Dean Michael Ancajas†, Koushik Chakraborty†, Sanghamitra Roy†
*Qualcomm Inc., San Diego, California
{saurabh.e.k}@gmail.com
†BRIDGE Lab, Electrical and Computer Engineering, Utah State University
dbancajas@gmail.com {koushik.chakraborty, sanghamitra.roy}@usu.edu

***Abstract*—Degradation of transistor parameter values due to Negative Bias Temperature Instability (NBTI) has emerged as a major reliability problem in current and future transistor generations. NBTI Aging of SRAM cell leads to a lower noise margin, thereby increasing the failure rate. The physical register file, which consists of an array of SRAM cells, can suffer from data loss, leading to system failure. In this paper, we explore a novel approach by investigating NBTI stress and mitigation at the instruction granularity. While a wide range of NBTI stress exists in different registers, the stress induced by specific instructions is highly predictable. Using such a prediction mechanism, we propose an NBTI tolerant power efficient physical register file design. Our approach improves the noise margin in a register file by 20%, 32%, and 125% for the 45nm, 32nm, and 22nm technology nodes, respectively. Overall, we observe 14.8% power saving and a 19.8% area penalty in the register file.**

I. Introduction

Negative Bias Temperature Instability (NBTI) has emerged as a major reliability challenge for the semiconductor industry in recent years. SRAM cells, which are the key elements in register files and caches, are severely affected due to NBTI aging. This degradation of reliability in the SRAM cell can result in loss of the stored value. Therefore, register files and caches are highly prone to failure due to NBTI.

In this paper, we use a novel approach to mitigate NBTI in the physical register file using an instruction-level analysis. We investigate NBTI stress from output values generated after instruction execution and its propagation in the register file. We observe that static instructions, uniquely identified by their *Program Counter (PC)*, frequently induce a predictable NBTI stress on the physical register file. Exploiting this program characteristics, we explore several predictor designs to detect instructions producing a large NBTI stress. Subsequently, we propose an NBTI tolerant physical register file design based on NBTI stress prediction.

Recent works have proposed techniques to improve reliability in the physical register file by extending the recovery period during idle cycles [1], [12]. They manipulate bit cell contents during the idle periods of physical registers to relax one or more PMOS transistors. The effectiveness of these techniques strongly depends on the length of the available idle period. A power-efficient physical register file, where its total capacity is closer to the architectural register file size, is likely to have a substantially shorter idle period, thereby limiting the reliability boost seen in these techniques. Kumar et al. investigate the SRAM reliability characteristics and propose the bit inversion technique for SRAM based caches [7]. However, their technique cannot be used for the register file as it may add extra delay for register access.

The main contributions of this paper are listed below.

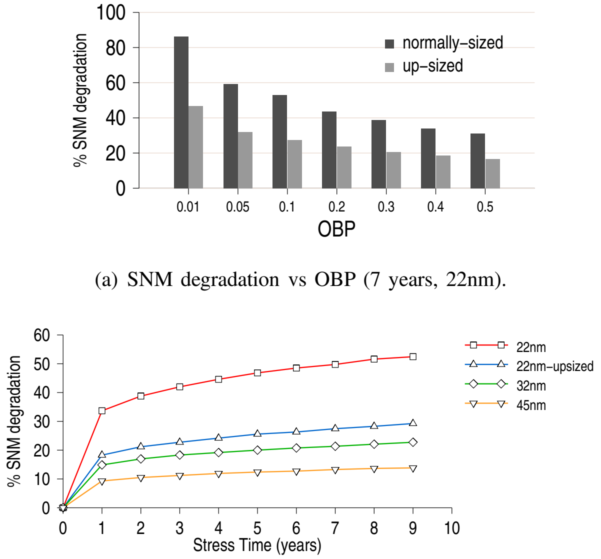
- We investigate the source of NBTI stress in a register file by studying output value patterns and their storage in a physical register file. We develop a mechanism to predict instructions producing high NBTI stress and evaluate various predictor designs for efficient prediction.
- We propose modifications in the physical register file by creating register banks and using sized transistors. Using larger sized transistors offers greater NBTI tolerance, at the cost of higher energy consumption. Our proposed technique exploits predictability of NBTI stress at the decode stage and predominance of narrow width values [3] to design a power-efficient NBTI tolerant physical register file. We modify the register renaming policy to incorporate NBTI-aware register allocation for our proposed design.
- Finally, we evaluate our proposed approach using a state-of-the-art full-system simulation infrastructure. We find that our approach improves the noise margin in a register file by 20%, 32%, and 125% for the 45nm, 32nm, and 22nm technology nodes, respectively. After accounting for all overheads, we observe a 14.8% power saving, while incurring a 19.8% area penalty.

The remainder of the paper is organized as follows: Section II describes NBTI impact in an SRAM cell. In Section III, we explain the link between instruction execution and NBTI stress. In Section V, we describe our proposed changes in the physical register file structure. Section VI gives details of our simulation infrastructure and experimental setup. In Section VII, we present results. We conclude in Section VIII.

II. Background

In this section, we estimate NBTI degradation in the SRAM cells. We study two primary factors in the extent of degradation: (a) input bias probability, (b) transistor size. In our analysis, we use the *Static Noise Margin (SNM)* as the measure of the robustness. SNM is calculated using the butterfly curve

978-1-4673-3052-7/12/$31.00 ©2012 IEEE



(a) SNM degradation vs OBP (7 years, 22nm).

(b) Lifetime SNM Degradation

Fig. 1: SNM Degradation in an SRAM cell due to NBTI. *up-sized* indicates SRAM cells built with wider transistors.

of SRAM cells [13]. The methodology used to measure NBTI impact is described in Section VI.

A. *Input Bias Probability*

The input pattern to a transistor is the primary driving factor for NBTI degradation. The effects of NBTI are caused by certain input values to the transistors. To understand the input pattern better, we introduce a parameter called One Bias Probability (OBP). OBP is the probability of a transistor input to be at logic 1. When the OBP is close to 0.0 (0%), the PMOS transistor is stressed for most of the time.

Figure 1(a) shows the SNM degradation of an SRAM cell for different OBP values. It can be observed that robustness improves as OBP is increased to 0.5. The SNM is degraded most when the SRAM cell undergoes high unbalanced stress (OBP closer to 0.0).

B. *Transistor Size*

Transistor sizing is one of the methods used to increase tolerance towards NBTI. Previous works have used transistor sizing for NBTI mitigation in combinational circuits [6], [14]. In our work, we exploit up-sized transistors to improve the SNM of an SRAM cell. The line marked with the up-triangles in Figure 1(b) indicates SNM degradation for such a cell. It can be observed that SNM degradation reduces by a large extent when compared to an SRAM cell without sized transistors (line marked with squares). Similarly, Figure 1(a) shows improvement in the SNM for all OBP values compared to the original cell.

III. Instructions and NBTI stress

In this section, we study the relationship between the bit patterns of values computed from individual instructions and its impact on NBTI stress. We find that instructions often have highly predictable NBTI stress patterns. This observation opens up new opportunities to design NBTI-aware microarchitectures.

A. *Bias Predominance*

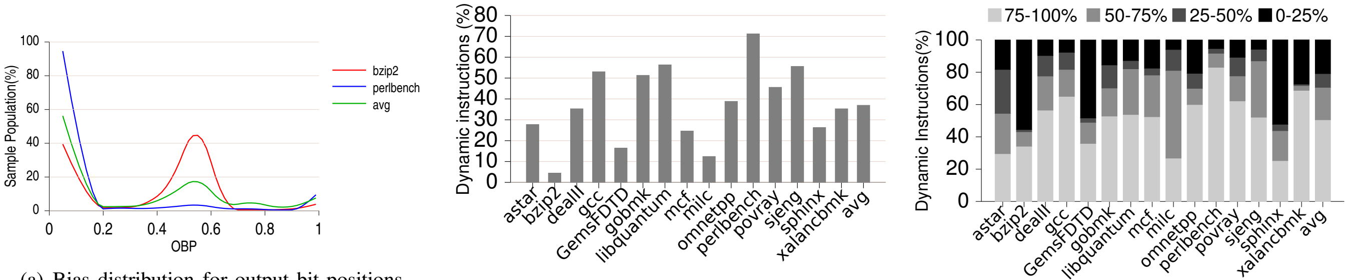
Typically, values stored in a register file are narrow-width values [8], [3]. Hence, there is a large number of bit positions at logic 0, which can potentially lead to a high NBTI stress. Similarly, there is a substantial number of instructions producing outputs capable of generating NBTI stress. To quantitatively measure the NBTI stress generated by an instruction, we look at the instruction's output bias. When a majority of output bits are at logic 0, the output can generate a high NBTI stress. We define a parameter to indicate the output's bias towards logic 0, *viz. Zero Predominance* (ZP). We formally define ZP as follows: The ZP of an instruction is 1 when more than 75% of its output bits are at logic 0. Similarly, we say that a *zero-predominant* instruction if its ZP is equal to 1. ZP indicates an instruction's ability to produce NBTI stress in the register file.

Figure 2(a) shows the presence of a large number of bit positions in a register file undergoing high NBTI stress. We plot the distribution curve of each bit position against its OBP. Figure 2(a) demonstrates bias distribution curves for two benchmark programs, widely varying in their characteristics. The perlbench has its highest peak near 0.0, meaning it has the largest number of bits at logic 0. On the other hand, bzip2 has the smallest number of bit positions at logic 0. Relatively, the number of bit positions at logic 1, is small for all programs. On average, 57% of the bit positions are always at logic 0 and 18% of bits have OBP close to 0.5. In other words, 57% of SRAM cells in a register file might store logic 0 during their entire lifetime. The observation above proves that a large number of SRAM cells in the register file can potentially suffer from unbalanced stress and hence high SNM degradation.

Figure 2(b) shows a plot of zero bias predominance for SPEC CPU2006 benchmark programs. Each bar indicates the percentage of zero-predominant instructions. Perlbench has the highest number of zero-predominant instructions, at 71.16%. On the other hand, bzip2 has the least number of zero-predominant instructions. On an average, 37% of the instructions have the potential to create high NBTI stress in a register file.

B. *Bias Predictability*

Figure 2(b) shows that a large number of instructions produce high NBTI stress in the register file. Detection of such instructions before execution can be used to efficiently handle NBTI stress. In this subsection, we discuss the predictability of instructions to generate a large stress in a register file. To predict the bias predominance of any dynamic instruction, we look at the output bias probabilities of a corresponding static instruction. When the OBP of a bit position is 0.0 or 1.0, it implies that the value has not changed during the entire execution time. Hence, the value of such a bit position is highly



(a) Bias distribution for output bit positions

(b) Zero Predominance of instructions that compute new value.

(c) Level of predictability for instruction outputs.

Fig. 2: Some instructions produce high NBTI stress in the register file. These instructions are predictable.

predictable. To measure the predictability of the complete instruction output, we count the predictable bits in the output. Based on the number of predictable output bits, we classify instructions into four categories.

Figure 2(c) plots the percentage of instructions with four levels of output bias predictability. Each bar in the plot is made up of four stacked components corresponding to different levels of the output bias predictability. The lowermost component indicates the set of instructions that produce output values with more than 75% of bit positions being predictable. In other words, more than 75% output bits of such instructions have an OBP close to 0.0 or 1.0. The second component from the bottom indicates the set of instructions that have more than 50% predictable output bit positions. On an average, 71% of dynamic instructions have more than 50% predictable bit positions. This observation proves that a large number of instructions that produce high NBTI stress can be predicted. Next, we develop a mechanism to identify and predict occurrence of stress-generating instructions.

IV. Predicting Instruction Level NBTI Stress

In this section, we discuss key predictor designs and analyze their performance. We explore three specific designs: (a) Last Value Predictor (Section IV-A), (b) Bimodal Predictor (Section IV-B), and (c) NP Predictor (Section IV-C).

A. Last Value Predictor (LVP)

Figure 2(c) shows that approximately 51% of the instructions produce values with constant bias. With this observation, we introduce the Last Value Predictor that predicts the zero predominance of the output to be the same as the previous instance. The LVP stores the previous ZP value in the history table for each static instruction. As the ZP can only have two possible values, a single bit for each static instruction will suffice.

The prediction accuracy of the LVP is highest when instructions produce constant or sequential values. For instance, *for-loop* counters will have the same bit values in its most significant bits, making them easily predictable. The LVP faces misprediction each time the instruction output predominance is different than the previous instance. Hence, a single deviation from a predominance pattern results in two mispredictions. Considering the above cases of mispredictions, we improve the predictor design by using two bits to store the predominance state.

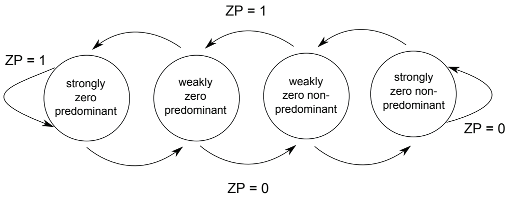

ZP = 1
ZP = 1
strongly zero predominant
weakly zero predominant
weakly zero non-predominant
strongly zero non-predominant
ZP = 0
ZP = 0

Fig. 3: Bimodal Predictor State Transition

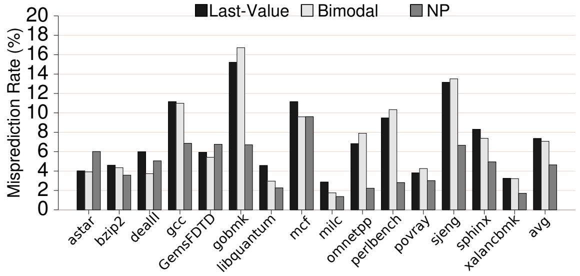


Fig. 4: The misprediction rate.

B. Bimodal Predictor (BP)

With the Bimodal predictor, rather than storing the actual ZP value in the history table, we store the state of bias predominance prediction. The state of bias predominance prediction is represented with the help of a 2-bit saturating counter. Figure 3 shows the state diagram for the BP. Initially, the counter is in the *strongly zero predominant* state. It remains in that state as long as the ZP of each instruction's instance is high. When an instance produces an output with a low ZP, its state is changed to *weakly zero predominant*. Successive instances of instructions with low ZP values will change the state to *strongly zero non-predominant*. The *Strongly zero non-predominant* state corresponds to the pattern where instances of the instructions produce non-zero-predominant output values.

The BP adapts better in situations where the ZP value is alternately changed. It can also tolerate a single deviation from a bias predominance pattern without misprediction. However, the BP results in two mispredictions before adapting to a new pattern.

C. NP Predictor

The LVP and BP predictors store the prediction state of all encountered instructions, regardless of their ZP value. The NP (Non-zero Predominance) predictor only keeps track of non-zero-predominant instructions. Hence, its predictor table is composed of the PC addresses of the instructions that are likely to produce non-zero-predominant outputs. As a limited number of instructions are tracked, fewer entries are stored in the NP predictor compared to LVP and BP. Consequently, this leads to increased coverage and lesser conflicts as more entries can be stored in the NP predictor's history table.

For the NP Predictor, a misprediction can occur in two ways: 1.) An instruction having an entry in the table produces a high ZP output. 2.) An instruction not having entry in the table produces a low ZP output. In this work, we only consider the first case because we find that it results in lower misprediction rates and lower complexity in the remapping and predictor circuitry. Considering the second case would mean tagging the instructions that are not in the table, and remapping their destination registers once they produce a low ZP output. This places unnecessary burden in the pipeline because unlike the former case, such instructions can continue to execute without affecting the correctness of the output value. However, we do update the predictor table so that the next time this instruction is encountered it will be correctly predicted. In sum, the second case only represents a lost opportunity in using the narrow-width registers.

D. Predictor Performance

Figure 4 presents the misprediction rates of the three predictors, each having 8k entries. It can be seen that the misprediction rate of the NP Predictor is the lowest among the three predictors. We find that majority of the misprediction stems from the conflict of entries in the predictor table. As such, the NP predictor yields lower misprediction rates since it only tracks instructions with low ZP output. Consequently, low ZP instructions are also highly predictable in nature. On average, NP misprediction rate is about 40% less compared to the other two predictors.

V. Minimizing NBTI degradation in the Register File

After successful prediction of instructions that generate a high NBTI stress, we use this information to minimize NBTI impact in the register file. In the following subsections, we describe proposed modifications in the physical register file for improving reliability.

A. Design Overview

We split the register array into two banks, each having an equal number of registers. One of the register banks is exclusively used for allocating registers for the outputs of zero predominant instructions. As most of the zero predominant outputs are narrow-width, we compress register widths to 16-bits, which is discussed more in Section V-B3. The reliability of the narrow-width register bank is further improved by using up-sized transistors. The second register bank is used for the remaining instructions, which produce non-zero predominant outputs.

We change the register allocation policy to allocate registers based on the predicted ZP value of an instruction. We handle special cases of mispredictions by performing a remapping of the physical registers. Sections V-C and V-D describe in detail the modifications in the decode and execute pipeline stages to handle mispredictions.

B. Register File Modifications

In this subsection, we describe our proposed changes in the structure of the physical register file.

1) Banked Register File: As mentioned earlier, the SRAM cells storing logic 0 (or 1) for a long period of time undergo high NBTI stress. The SRAM cells of the more significant bits in a register file show a similar behavior. Compressing such outputs can reduce the number of bits at logic 0, without losing important information. Based on this observation, we divide the register array into two banks with different register widths. Configuration (c) in Figure 5 shows the new register file with two banks of variable widths. The first bank consists of 64-bit registers, while the second bank consists of compressed 16-bit registers.

2) NBTI Tolerant Bank: By compressing the widths of zero predominant outputs, we get rid of a large number of bits storing logic 0 and save a substantial amount of NBTI stress. However, there could still be many bits at logic 0 in the compressed output. These bit positions can further reduce the overall reliability of a physical register file. To increase the noise margin of a narrow-width bank, we use up-sized transistors for NBTI tolerance. As seen in Figure 1, the effective SNM degradation of SRAM cells employing up-sized transistors is smaller than that of SRAM cells with normally sized transistors. Therefore, the introduction of up-sized transistors improves the overall reliability of entire register file.

Configurations (b) and (d) in Figure 5 show tolerant register banks in gray. Benefits of register file banking and up-sized transistors are discussed in Section VII. Next, we describe the modified register allocation policy and special cases of mispredictions in detail.

3) Register Compression: In modern 64-bit processors, most register outputs that are generated do not require the full width of the 64-bit registers. Typically, these output values have leading 0's or 1's that can be truncated. We formally refer to these contents as *narrow-width* values because they can be stored in a register with fewer bits. We apply compression only on narrow-width values because they are simple to compress and hence, will have low hardware overhead.

Before compression is to be done, it must be determined if an output value is *narrow-width* or not. We insert a detector circuit after each functional unit that will signal if an output is *narrow-width.* This circuit can be implemented as a combination of basic logic gates (AND, NOT, OR) to compare the most significant bits in the outputs.

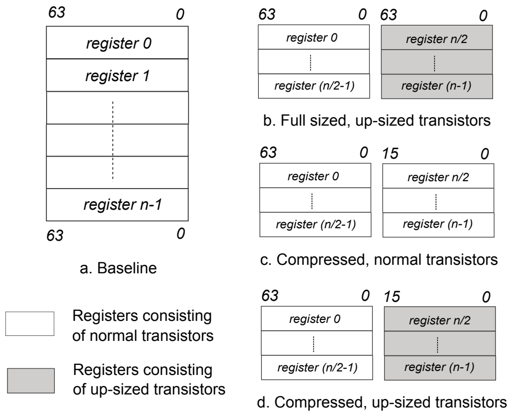
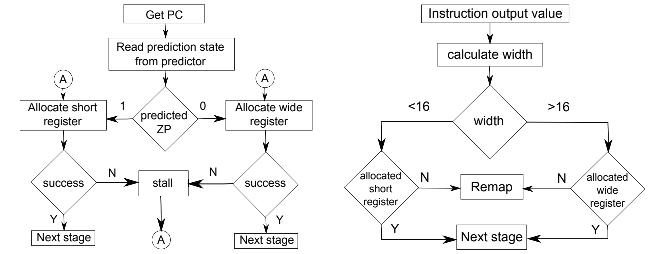

63
0
register 0
register 1
register n-1
63
0
a. Baseline
register 0
register (n/2-1)
register n/2
register (n-1)
b. Full sized, up-sized transistors
15
c. Compressed, normal transistors
d. Compressed, up-sized transistors
Registers consisting of normal transistors
Registers consisting of up-sized transistors

Fig. 5: Various configurations of the register file.



(a) Flow of events during decode stage

(b) Flow of events during execution stage to handle bank mispredictions

Fig. 6: Modifications in Register Allocation Policy.

C. Modified Register Allocation Policy

We require modifications in the register allocation policy as one of the banks is exclusively used for zero predominant outputs. Figure 6(a) outlines the new register allocation process. After predicting the ZP value for a given instruction, a register from one of the banks is allocated. If the ZP is high, a short register from the narrow-width bank is allocated as its destination. Similarly, a wide register is allocated for a non-zero-predominant instruction. During certain program phases, when there is a high demand for registers from one of the banks, register allocation may fail. In such situations, we stall the given instruction in the decode stage and wait for the registers to be free.

D. Handling Mispredictions:Remapping in Execution Stage

Figure 6(b) shows the sequence of events during a misprediction. After the execution unit has calculated the output, its width is calculated to determine if an appropriately-sized register has been allocated. If the output is a non-narrow value, a wide register must be used as its destination. If the register allocated in the earlier stage is a narrow one, then the execution stage is stalled to allow for remapping.

The remapping of physical registers in the execution stage is done by modifying the rename table and updating the instruction register dependencies. Remapping is accomplished in two steps: 1.) A free destination register must be allocated. 2.) Instructions in the earlier pipeline stages that are dependent on the mispredicted instruction must also update their source register mappings. This is done by broadcasting both the old and new tags of the renamed register so that the issue queue entries are fixed in-place.

Simply stalling the fetch and decode stage when remapping, as done during unavailability of instruction window entries or free physical registers, will not be sufficient because it can lead to a pipeline deadlock when there are no free registers. To handle this case, we squash younger instructions from the pipeline and update the predictor table once we see that the physical register is full. Squashing younger instructions adds some registers in the free pool, which can be used by the mispredicted instruction once it is re-executed.

E. Performance Overhead

With the modified register file, performance of the processor can suffer in three different situations:

1) During the decode stage, performance is lost if there are no free registers in the requested register bank. The instruction is stalled until an older instruction commits and frees up a register.
2) During remapping in the execute stage, we assume a one clock cycle penalty to correct the register mapping, given that there is a free register.
3) When there are no free registers during remapping in the execute stage, the younger instructions in the pipeline need to be flushed, as explained earlier. We assume a four cycle penalty to re-execute the mispredicted instruction. Though this only happens rarely in modern processors as more than 95% of the time, only 50% of the physical register file capacity is used [4].

We find that the loss in IPC due to the above cases are on average 1.75%. We discuss the results of performance overhead in Section VII-C.

VI. Methodology

A. Architectural Simulations

For investigating register file configurations and its impact on performance, we use a full-system simulator built on top of the Wind River SIMICS [9]. For our experiments, we use the SPARC V9 ISA. However, we use our own detailed timing model to enforce timing characteristics of a 4 wide superscalar out-of-order core. Our modeled processor has a register file structure similar to SPARC V9. The architecture register file contains 160 windowed registers. We have implemented MIPS R10K style register renaming [15] with a physical register file of 224 registers. Various register file configurations shown in Figure 5 are explained in section VII.

We use several SPEC CPU2006 benchmarks on a Solaris 9 [5]. We use the three most representative Simpoint[11] phases from these SPEC benchmarks in our study.

B. NBTI Effect Measurement

When the input to a PMOS transistor is low, holes in the inversion layer break Si-H bonds at the oxide layer. This phenomenon leads to an increase in the absolute value of the threshold voltage of the transistor (*stress phase*). When the input to a PMOS transistor is high, the threshold voltage slowly starts restoring to the original value (*recovery phase*). We model a standard 6-T SRAM cell in HSPICE to measure NBTI impact. The NBTI wearout alters the transfer characteristics of cross-coupled inverters, decreasing the noise margin. We evaluate the SNM at different time intervals by measuring this potential difference. We use a predictive model to determine the change in transistor threshold voltage due to NBTI [2].

C. Methodology for Area and Power Estimations

For finding the savings in Area and Power of the modified register file, we describe structures in Verilog. We synthesize the hardware using the Synopsys Design Compiler and the 45nm TSMC library. We synthesize various register file configurations for fixed latency and obtain the area and power estimation from this synthesized Verilog. We use the CACTI 6.0 tool to perform area and power analysis of the predictors [10] for the 45nm technology node.

VII. Results

In this section, we present the results of improvement in SNM, power efficiency and performance overhead due to modifications in the register file. We compare our results with the *Recovery Boosting* technique discussed in [12]. We experiment with various configurations of the physical register file and show the results for SNM improvement in Figure 7. Then we discuss the savings in area and power due to our proposed technique. Finally, we show performance impact of our proposed schemes.

We evaluate six different configurations of the register file, and present results with respect to the *Baseline* configuration.

- *Baseline:* This refers to the traditional design of a physical register file with 64-bit wide registers only. Refer to configuration (a) in Figure 5.
- *comp,norm:* This refers to a compressed banked register file design where one of the banks has 16-bit wide registers only. Refer to configuration (c) in Figure 5.
- *rec:* This refers to the modified register file in [12] using the *Recovery Boosting* technique during the idle period. In our processor configuration, we found the idle period to be in the range of 16.65%-58%, with an average of 45.27% of the total physical register lifetime.
- *full,up:* This refers to a banked physical register file where all registers are 64-bit wide only. One of the banks is made up of up-sized transistors. Refer to configuration (b) in Figure 5.
- *comp,up:* This refers to a compressed banked register file design where one of the banks has 16-bit wide registers only. One of the banks is made up of the up-sized transistors. Refer to configuration (d) in Figure 5.

A. Results for SNM improvement

Figure 7 shows the average percentage improvement in the SNM of the register file configurations for the 22nm, 32nm and 45nm technology nodes. The first bar in the cluster indicates 9-12% (32nm) improvement in SNM with compression of one register bank to 16-bits. The *Recovery boosting* technique results in 18-26% SNM improvement. For *full,up*, the SNM increase is highest among all configurations. The last configuration *comp,up* gives a smaller improvement in SNM compared to *full,up*, but with the highest savings in power and area. Ideally, we expected the SNM to increase from *full,up* to *comp,up* as the number of SRAM cells of a higher significant bits are reduced. But after careful analysis, we found that the SNM of the significant SRAM cells with up-sized transistors is more than the average SNM of the first bank. Hence, SRAM cells with up-sized transistors for significant bits improve the average SNM of the entire register file. On an average, *comp,up* shows an SNM improvement of 20%, 32%, and 125% for the 45, 32 and 22nm technology nodes, respectively.

B. Area and Power Comparison

Our technique improves SNM by compressing registers without losing the functional correctness. Effectively, the modified register file has fewer SRAM cells. Reducing the size of hardware results in savings in area and power. Figure 8(a) shows the percentage savings for area and power for our proposed techniques with respect to the *baseline*.

Configuration *comp,norm*, where the second register bank is compressed to 16-bit only, provides 34.5% savings in the overall area. As the number of SRAM cells are reduced, both dynamic and leakage power decrease. Overall savings in dynamic, leakage and total power are 37%, 29.9% and 34.4%, respectively.

When transistors are up-sized to give a better SNM, they consume more power and occupy a larger area. Hence, up-sizing the transistors in SRAM cells results in area and power overhead. For *full,up*, overall area and power increases by 11.2% and 5%, respectively. In *comp,up*, area and power overhead is compensated by reducing the width of one of the register banks. Effective savings in area and power are 28% and 33%, respectively.

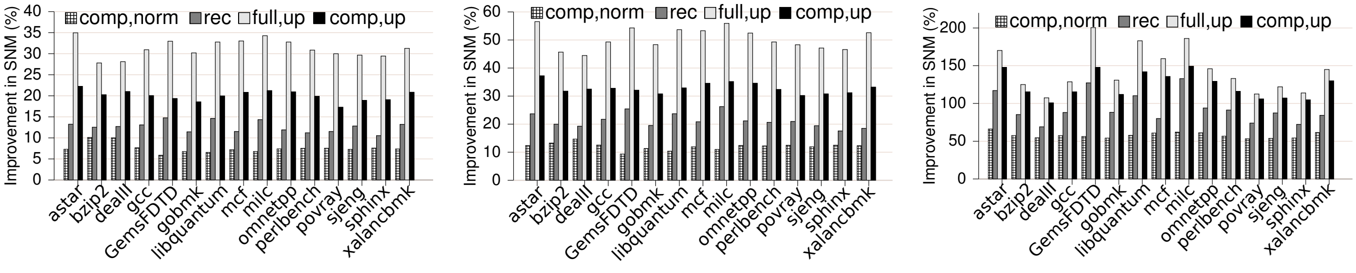
Figure 8(b) shows the combined area and power savings for the technique *comp,up* along with the NP predictor compared to the *Baseline*. We can see that the total power savings due to our proposed approach varies from 30.68% to 14.86% as the predictor size is increased from 512 to 8k. We save area in all configurations except for the 8k predictor. Configuration *comp,up* combined with the 8k predictor has an area overhead of 19.77%.

C. Performance Analysis

Figure 9 shows the performance impact due to mispredicted instructions being stalled, remapped and sometimes causing pipeline flushes. The average IPC reduction across all benchmarks is only 1.75%. *Gobmk* and *mcf* experience the most reduction because they have the highest misprediction rates. *xalancbmk* and *milc* observe the least performance reduction due to their low misprediction rates.

VIII. Conclusion

NBTI is one of the critical challenges for the semiconductor industry today. NBTI in the register file reduces the overall

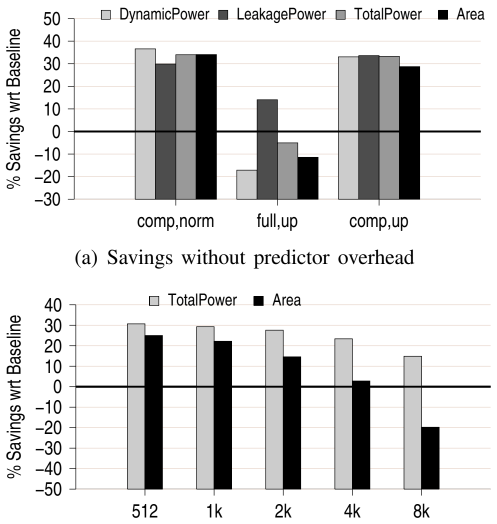


(a) 45nm

(b) 32nm

(c) 22nm

Fig. 7: Percentage improvement in SNM with respect to the *Baseline*.



(a) Savings without predictor overhead

(b) Savings with predictor overhead

Fig. 8: Area and Power savings for various configurations of the proposed register file.

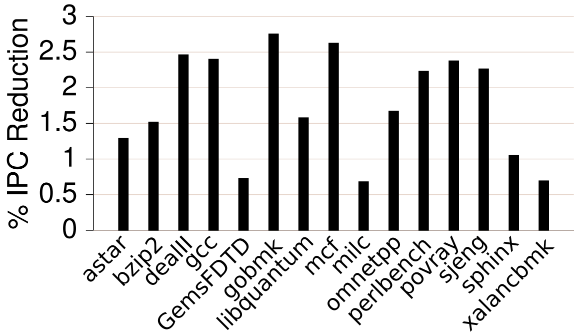


Fig. 9: Performance Overhead due to Stalling, Remapping and Pipeline Flushing.

reliability of the processor and potentially causes system failure. In this paper, we proposed a novel approach for mitigating NBTI by looking at instructions producing a large NBTI stress in a register file. We found that some of the instructions produce values that can cause high NBTI stress in the register file, while other instructions produce values which have limited impact. With this observation, we designed a prediction mechanism for detecting instructions that produce a high NBTI stress. To increase the overall reliability of the register file, we divided the register file into two banks and used up-sized transistors. We show that this approach provides substantial improvement in the average SNM of the register file. Our techniques also result in reduction of total area and power consumption.

Acknowledgments

This work was supported in part by National Science Foundation grant CNS-1117425 and Micron Foundation. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] J. Abella, X. Vera, and A. González. Penelope: The nbti-aware processor. In *Proc. of MICRO*, pages 85–96, 2007.

[2] S. Bhardwaj, W. Wang, R. Vattikonda, Y. Cao, and S. Vrudhula. Predictive modeling of the nbti effect for reliable design. In *IEEE Custom Integrated Circuits Conference*, pages 189 –192, sept. 2006.

[3] D. Brooks and M. Martonosi. Dynamically exploiting narrow width operands to improve processor power and performance. In *HPCA*, pages 13–22, 1999.

[4] J. A. Butts and G. S. Sohi. Use-based register caching with decoupled indexing. In *Proceedings of the 31st annual international symposium on Computer architecture*, ISCA '04, pages 302–, Washington, DC, USA, 2004. IEEE Computer Society.

[5] J. L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, 2006.

[6] K. Kang, H. Kufluoglu, M. Alain, and K. Roy. Efficient transistor-level sizing technique under temporal performance degradation due to nbti. In *International Conference on Computer Design*, pages 216 –221, oct. 2006.

[7] S. V. Kumar, C. H. Kim, and S. S. Sapatnekar. Impact of nbti on sram read stability and design for reliability. In *Proc. of ISQED*, pages 210–218, 2006.

[8] M. Lipasti, B. Mestan, and E. Gunadi. Physical register inlining. In *Proc. of ISCA*, pages 325 – 335, june 2004.

[9] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, Feb 2002.

[10] N. Muralimanohar, R. Balasubramonian, and N. Jouppi. Cacti 6.0: A tool to model large caches. *School of Computing, University of Utah, Technology Report*, 2007.

[11] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *PACT*, pages 3–14, 2001.

[12] T. Siddiqua and S. Gurumurthi. Enhancing nbti recovery in sram arrays through recovery boosting. *IEEE Trans. on VLSI Systems.*, PP(99):1, 2011.

[13] N. Weste and D. Harris. *CMOS VLSI Design: A Circuits and Systems Perspective*. Addison Wesley, 2004.

[14] X. Yang and K. Saluja. Combating nbti degradation via gate sizing. In *Proc. of ISQED*, pages 47 –52, march 2007.

[15] K. Yeager. The mips r10000 superscalar microprocessor. *IEEE Micro*, 16(2):28–41, April 1996.