NBTI-Aware Dynamic Instruction Scheduling

Taniya Siddiqua and Sudhanva Gurumurthi
Department of Computer Science
University of Virginia
Charlottesville, VA 22904
{taniya, gurumurthi}@cs.virginia.edu

Abstract—**NBTI is an important emerging silicon reliability problem. In this paper we explore a microarchitecture-level approach to mitigate NBTI related failures in the functional units of a superscalar processor. We analyze the impact of dynamic instruction scheduling on NBTI and show that the conventional approach to instruction scheduling can accelerate the wear out of certain functional units. We then propose and evaluate two different NBTI-aware instruction scheduling policies and quantify the tradeoffs between performance and reliability of each policy.**

I. Introduction

With continued technology scaling, processors are becoming more susceptible to transient faults and hard errors. Hard errors or hard failures are permanent faults that occur due to the wearing out of various hardware resources over time as a result of their usage. These failures are dependent on design-time factors such as process parameters and wafer packaging, as well as runtime factors such as temperature and resource utilization, which depend on the execution characteristics of the workload. Some well known hard error phenomena include Electromigration (EM), Thermal Cycling (TC), Time Dependant Dielectric Breakdown (TDDB), Stress Migration (SM) and Negative Bias Temperature Instability (NBTI). Designing processors that can operate reliably in the presence of these different fault phenomena is a key challenge facing the microprocessor industry.

There has been recent research on addressing lifetime reliability issues via Dynamic Reliability Management (DRM) by using techniques such as Dynamic Voltage/Frequency scaling [20] and also using adaptive supply voltages and body biasing techniques at the granularity of entire processor cores [8]. However, there are opportunities to manage lifetime reliability even *within* a core by careful management of the hardware resources. The usage characteristics of different on-chip hardware resources depend on microarchitectural factors such as instruction scheduling policies and instruction fetch policies (in the case of a multi-threaded processor). Moreover, in multicore processors that use simple cores (e.g., IBM POWER6, Intel Nehalem, SUN Niagara), it is important to manage the hardware resources within each core carefully so that one can extract high performance from each core over the lifetime of the entire chip. In this paper, we explore the idea of intra-core reliability management and analyze the impact of dynamic instruction scheduling on one important emerging lifetime reliability problem - NBTI. We choose to study instruction scheduling since it affects the utilization (and hence the lifetime reliability) of the functional units in the processor. Towards this end, this paper makes the following two contributions:

- We evaluate the impact of a conventional non-replay based dynamic instruction scheduling policy on NBTI related failure of functional units. We show that this scheduling policy leads non-uniform usage of the functional units and excessive NBTI related wearout of a subset of the functional units.
- We then propose and evaluate two different NBTI-aware scheduling policies in terms of their performance and reliability impact.

There have been prior works on developing reactive techniques to handle hard errors and/or manage reliability at the granularity of entire cores [8]. Our goal in this research is to develop techniques that can *proactively* manage NBTI at the granularity of individual or groups of microarchitectural resources *within* the core. A recent work has motivated a proactive approach to NBTI management of SRAM cells in the cache [18]. The goal of our research is to explore extending this reliability management approach to other structures in the processor.

II. Related Work

In this section we present an overview of key related research on lifetime reliability. Yilmaz et al [1] present a lazy error detection technique for the functional units, which takes advantage of the delay between the execution and commit time of instructions that produce data to detect errors. Bower et al [2] propose a microprocessor design integrated with DIVA-style dynamic verification based error detection mechanism [4] to diagnose hard faults. After diagnosis it deconfigures the faulty unit to continue fault-free operation. Meixner et al [3, 5] use flow verification to verify the operational correctness of the processor. Schuchman and Vijaykumar [26] employ redundant multithreading [7] to detect hard errors where trailing thread instructions do not use the same pipeline resources used by the corresponding leading instructions. In all these efforts, a reactive approach is taken to make a processor fault-tolerant to hard errors. Our paper proposes a proactive approach to reliability.

III. Overview of Dynamic Instruction Scheduling

The dynamic instruction scheduler decides which instructions are executed and at what times on a given set of functional units. The scheduler consists of two key components: wakeup logic and select logic. The wakeup logic is responsible for asserting an instruction as being 'ready' in the issue window by updating source dependencies of instructions waiting for their source operands to become available. Every time a result is produced by a functional unit, the tag of the result is broadcast to the waiting instructions in the issue window. Each waiting instruction compares the result tag with the tag of each of its source operands. Once both operand tags have matched, the instruction is ready to execute (*instruction wakeup*) and the ready instructions signal the select logic to request execution on a given type of functional unit. Once a functional unit becomes available, the select logic directs a suitable instruction to that unit for execution by asserting the corresponding grant signal (*instruction select*). Since multiple instructions could wakeup in a given cycle and the processor typically has multiple functional units of the same type, there needs to be a policy to select a subset of the ready instructions and assign them to specific functional units based on resource availability. Modern instruction schedulers typically use a form of prioritized scheduling where instructions are selected in an oldest-first order from the issue window and each instruction is issued to the lowest-numbered available functional unit. For example an instruction will be allocated to FU_0 if it is available; if FU_0 is busy, then FU_1 will be checked for availability and so on. This non-uniform assignment leads to the case where functional units with smaller sequence numbers tend to get utilized more and hence experience a higher degree of wearout. Although there has been prior work on providing graceful degradation by detecting and deconfiguring faulty components inside the processor [1, 2, 3, 5, 21], the permanent loss of a functional unit could have serious repercussions on performance. The performance impact would be especially pronounced for multicore processors that use simple cores that have a relatively narrow pipeline width.

IV. NBTI

Negative Bias Temperature Instability (NBTI) is becoming a growing concern for CMOS technologies and affects the lifetime of PMOS transistors. NBTI increases the threshold voltage of PMOS devices, which in turn degrades the speed of circuits. The switching delay (T_{sd}) of a transistor can be calculated using the alpha power law [23] where α=1.3 as:

$$T_{sd}\,\alpha\,\frac{V_{dd}L_{eff}}{\mu(V_{dd}-V_t)^{\alpha}}$$

When a logic input of '0' is applied to the gate of a PMOS transistor ($V_{gs}=-V_{dd}$), NBTI occurs due to the generation of interface traps at the Si/SiO2 interface. This is known as the *Stress* phase for the PMOS. The increase in V_t due to stress is given by the equation [17]:

$$\Delta V_{ts}=(\frac{qt_{ox}}{e_{ox}})^{\frac{3}{2}}.K_1.\sqrt{C_{ox}(V_{gs}-V_t)}.e^{\frac{-E_a}{4kT}+\frac{2(V_{GS}-V_t)}{t_{ox}E_{01}}}.T_0^{-0.25}.t_{stress}^{0.25}$$

where t_{stress} is the time under stress, t_{ox} is the oxide thickness and C_{ox} is the gate capacitance per unit area. K_1, E_a, T_0, E_{01} and k are constants equal to 7.5 $C^{-0.5}nm^{-2.5}$, 0.49 eV, 10^{-8} s/nm^2, 0.08 V/nm and 8.6174 × 10^{-5}eV/K respectively [17].

When a logic input of '1' is applied to the gate (V_{gs} = 0), the transistor turns off eliminating some of the traps. This is known as the *Recovery* phase. The final increase of V_t after considering both the stress and recovery phases is:

$$\Delta V_t=\Delta V_{ts}.(1-\frac{2\xi_1 t_{0x}+\sqrt{\xi_2 e^{\frac{-E_a}{kT}}T_0 t_{rec}}}{(1+\delta)t_{ox}+\sqrt{e^{\frac{-E_a}{kT}}(t_{stress}+t_{rec})}})$$

where t_{rec} is the time under recovery, ξ_2, ξ_1 and δ are constants equal to 0.5, 0.9 and 0.5 respectively [17].

Some recent papers have explored the effect of NBTI on a single functional unit [6, 19]. Based on the observation that functional units stay idle for a significant amount of time, these prior works propose vectoring certain 'special inputs' to these functional units to put them in recovery mode. In this paper we assume that an idle functional unit is always put in recovery mode using such input vectoring techniques.

V. Instruction Scheduling Policies

We now present a brief overview of the instruction scheduling policies that we evaluate. We start with the non-NBTI aware policy, which is our baseline, followed by our two proposed NBTI-aware policies. The goal of the NBTI-aware scheduling policies is to extend the recovery times for the functional units so that the lifetime degradation of functional units due to NBTI can be minimized.

A. Prioritized Scheduling (PS)

Prioritized scheduling is our baseline non-NBTI aware scheduling policy that we discussed in section III. In this policy, whenever an instruction becomes ready for execution, the select logic allocates the lowest numbered functional unit, if available, to the instruction. Otherwise the next lowest numbered free functional unit is allocated. To reduce the complexity of the select logic design, functional units are statically prioritized and hard wired to the select logic with that fixed priority [13, 22].

B. Priority Rotation Scheduling (PR)

The *PR* policy is geared towards achieving a balanced utilization of the functional units. This policy modifies the *PS* policy so that the priorities of the functional units are changed, in a round-robin fashion, after a fixed number of cycles ($Cycle_{PR}$) have elapsed. In the PR policy, we start with FU_0 having the highest priority and assign lower priorities to the other functional units based on their sequence numbers (i.e., FU_0 initially has the highest priority whereas FU_{n-1} has the lowest). After $Cycle_{PR}$, FU_1 gets the highest priority, FU_2 gets the second highest and so on and FU_0 has the lowest priority. The select logic is similar to the *PS* policy but with added functionality to change the priorities of the functional units after $Cycle_{PR}$ cycles.

C. Time-Dependent Scheduling (TD)

Time-Dependent Scheduling extends the *PS* policy to include an explicit fixed recovery period. In the *TD* policy, whenever the wakeup logic flags an instruction to be ready, the select logic allocates the lowest numbered functional unit that is available. After a functional unit is used, no other instruction is assigned to that particular functional unit for a fixed number of cycles ($Cycle_{TD}$). This policy allows the functional unit to undergo recovery for $Cycle_{TD}$ cycles after each stress phase. We can implement this in hardware by keeping the busy signal of the functional unit asserted for $Cycle_{TD}$ cycles after the functional unit is used. However, since the functional unit cannot be used during this time, there could be a detrimental impact on performance.

VI. Experimental Setup

We simulate two different types of processor cores: a 4-wide issue core and a simpler 2-wide issue core. The use of simpler cores is gaining popularity in multicore chips due to their relatively low power consumption. However, since the 2-wide core has fewer functional units, hard errors in those units can significantly affect their ability to exploit instruction-level parallelism and, more importantly, could accelerate the failure of the entire core due to the availability of fewer resources for providing graceful degradation. The parameters of our processor core models are given in Table 1.

Our experiments are carried out for the 32 nm process technology. We use the SimpleScalar 3.0 simulator [10] with Wattch [11] for modeling performance and power and HotSpot [12] for simulating the temperature behavior. We present results for eight benchmarks from the SPEC CPU2000 benchmark suite [14]. We choose these eight benchmarks based on the study by Phansalkar et al. that shows that these workloads (*crafty, applu, fma3d, gcc, gzip, mcf, mesa, twolf*) are representative of the entire SPEC2000 benchmark suite [16]. The benchmarks were compiled for the Alpha ISA and use the reference input set. We perform detailed simulation of each benchmark for the first 100-million instruction SimPoints [15].

In this paper, we focus only on the impact of NBTI on the integer ALUs. Although we consider both integer and floating-point benchmarks in our evaluations, several of the floating-point benchmarks have a considerable number of integer instructions in their instruction mix and therefore utilize the integer ALUs. Extending our approach to other functional units is part of our future work.

Technology Parameters		
Process Technology	32 nm	
Supply Voltage/Frequency	0.9 V/3 GHz	
Ambient Temperature	45 C	
Processor Parameters		
Pipeline Width	4	2
Fetch-Queue Size	16	8
Branch Predictor Type	Bimodal, 2K-entry	Bimodal, 1K-entry
RAS/BTB Size	64/2K-entry 4-way	64/1K-entry 4-way
Branch-Misprediction Latency	7	7
RUU Size/LSQ Size	64/32	32/16
(Int/FP) ALUs/ (Int/FP)Multipliers	4/2	2/1
L1 Cache ports	4	2
L1 D-Cache/I-Cache	32 KB 4-way	16 KB 4-way
L2 Unified Cache	256 KB 4-way	256 KB 4-way
I-TLB/D-TLB	128 entries 4-way/ 256 entries 4-way	64 entries 4-way/ 128 entries 4-way
TLB Miss/ Main Memory Latency	30/100	30/100

Table 1: Processor Configurations Simulated. The latencies are given in terms of processor clock cycles.

Since NBTI affects PMOS transistors, it is necessary to have a circuit-level model of the functional unit. In this paper, we model the ALU as a Kogge-Stone adder for the purpose of estimating NBTI. We conservatively assume that all the PMOS transistors in the adder are affected equally by NBTI and calculate the output delay of the critical path of the adder. To compute the effect of NBTI on the ALUs due to different scheduling policies, we track the stress and recovery cycles of the functional units in the simulation and extrapolate these statistics to represent a full-day worth of activity and generate a utilization waveform of each ALU over that day. We update V_t every 86400 seconds and calculate the effect of NBTI after 7 yrs. We chose this period for updating V_t based on discussions with silicon reliability researchers at Intel. We use a $Cycle_{PR}$ value of 10K and a $Cycle_{TD}$ value of 2 for *PR* and *TD* policy respectively.

VII. Results

We first present results for the 4-wide issue core and then for the 2-wide issue core. For each core we present the NBTI impact of the baseline scheduler (*PS)* on the ALUs. We then present results for the other two scheduling policies. We use two metrics in our evaluations: lifetime of the ALUs and performance. We quantify the lifetime of an ALU as a percentage delay in output with respect to the original delay. We calculate the degradation in the threshold voltage from the simulation and the delay in output based on the model presented in Section IV.

A. 4-wide Issue Core

Figure 1 presents the effect of the *PS* policy on ALU lifetimes for the 4-wide issue core. The four ALUs of this core are denoted as ALU_0-ALU_3 respectively. After 7 years, ALU_0 experiences an output delay in the range of 5%-15% whereas

ALU3 experiences only a 3%-10% change in delay. This result clearly supports our hypothesis that the baseline instruction scheduler adversely impacts the lifetime of certain functional units, thereby accelerating their wear out.

In order to understand how scheduling impacts NBTI, we need to analyze how the functional units are utilized. Two factors influence the utilization, namely, the instruction mix and instruction level parallelism. The instruction mix gives an indication of how frequently each type of functional unit gets accessed. For example, the higher the number of integer instructions, the higher is the probability of accessing integer ALUs. Instruction level parallelism also determines the frequency of use of the functional units. Also, it affects how groups of functional units get used. When the IPC is high, more instructions are executed per cycle. Consequently, more functional units will get utilized. We profile the benchmarks to determine their instruction mix and instruction level parallelism. The instruction mix of the benchmarks, broken down into integer and floating-point instructions, along with their IPC are given in Table 2. Instructions that do not exercise the functional units are not shown in the table.

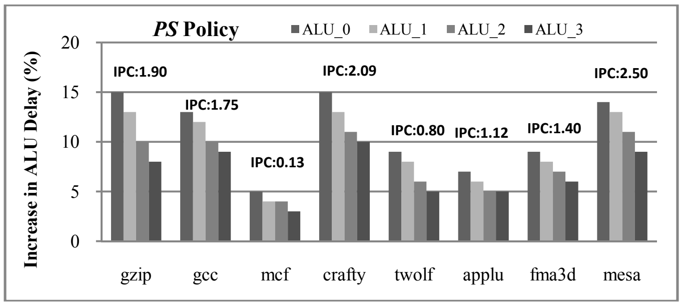

Figure 1: Degradation in ALU lifetime due to the *PS* Policy in 4-wide core

Benchmarks	Instruction Mix (%)		IPC
	Integer Instructions	Floating-point Instructions	
gzip	88.27	0.0	1.9
gcc	86.65	0.0	1.75
mcf	79.0	1.06	0.13
crafty	88.5	0.1	2.09
twolf	82.98	4.81	0.80
applu	17.1	82.22	1.12
fma3d	45.08	50.85	1.40
mesa	72.29	18.95	2.50

Table 2: Percentage of Integer and Floating-point instructions and IPC in the benchmarks

gzip and *crafty* have the highest percentage of integer instructions. Figure 1 shows that these two benchmarks experience the highest degradation as well. Since *crafty* has a higher IPC than *gzip*, the number of ALUs utilized per cycle should be higher for *crafty*. Consequently, all ALUs for the *crafty* benchmark experience equal or higher delay degradation compared to *gzip*. The least delay degradation is observed for *mcf* due to its very low *IPC*. Among the floating-point benchmarks, *mesa* experiences the highest degradation in lifetime reliability because this workload has the highest mix of integer instructions. Since it has the highest IPC with a large number of integer instructions (~73%), the delay degradation is very close to that of *gzip* and *crafty*.

The impact of NBTI due to the *PR* policy is presented in Figure 2. As intuition suggests, we can see that this scheme balances the lifetime of all the ALUs over the 7-year period. This policy improves the lifetimes of ALU0 and ALU1. On average ALU0 achieves a 23% improvement and ALU1 achieves 14% improvement in delay with respect to the *PS* policy, but also worsens the lifetime of the other functional units. Again, we see that the lifetime of the ALUs is a function of instruction mix and IPC. *crafty* has the highest integer instruction mix along with a high IPC, which leads to the highest degradation of the ALUs. Similarly, *mcf*, which has the lowest IPC, experiences the least amount of degradation.

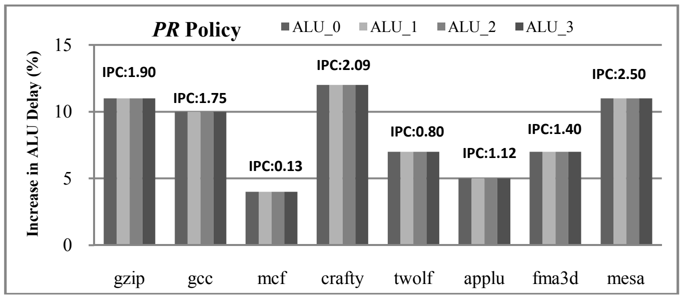

Figure 2: Degradation in ALU lifetime due to the *PR* Policy in 4-wide core

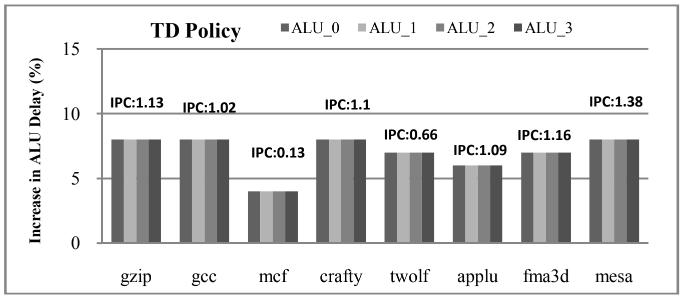

Figure 3: Degradation in ALU lifetime due to the *TD* Policy in 4-wide core

Figure 3 shows the effect of the *TD* policy. Among all the policies, *TD* is the most effective policy to improve the lifetime reliability of the ALUs. Every ALU gains some lifetime improvement, with ALU0 gaining a 33% delay improvement on average across all the benchmarks. However, this improved reliability comes at the cost of a reduction in performance. On average, the IPC drops by 26%. Benchmarks with higher IPC experience greater improvement in their ALU lifetimes. *gzip*, *crafty* and *mesa* gain almost a 50% improvement in the lifetime of ALU0. On the other hand, benchmarks with lower IPC do not suffer as much performance loss due to their inherently low instruction level parallelism. For example, *mcf* does not suffer a performance loss at all due to its very low IPC.

B. 2-wide Issue Core

Since the 2-wide issue core has only two ALUs, we would expect less variation in the utilization of the ALUs even for the baseline *PS* scheduling policy. Therefore it would have fewer opportunities for recovery. This trend can be seen in Figure 4 where the *PS* policy degrades the delay of ALU_0 in the range of 5%-15% and ALU_1 from 4%-12% with maximum delay difference of only 3%. Similar delay degradation behavior due to the instruction mix and the IPC is observed for 2-wide issue cores as well. *gzip* which has the highest integer instruction mix (~89%) and high IPC, experiences the greatest degradation in delays and *mcf* experiences the least amount of degradation.

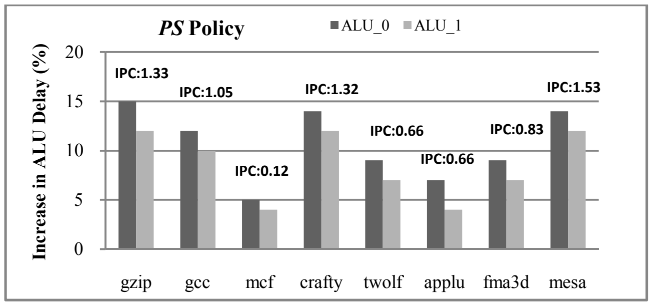

Figure 4: Degradation in ALU lifetime due to the *PS* Policy in 2-wide core

As Figure 5 indicates, the *PR* policy is not effective in improving reliability for the 2-wide core and achieves an overall delay improvement of only 12% for ALU_0 compared to the *PS* policy. For the 4-wide issue core, the IPC of the benchmarks is less than two most of the time, thereby leading to a lower ALU utilization. Therefore, there are more chances to offload the instructions to other ALUs, which provide better opportunities for the recovery for each ALU. But for a 2-wide core, this is not the case. The IPC values show that there are fewer opportunities to offload instructions to idle ALUs. Therefore, there is less scope for recovery and the load balancing attempted by the *PR* policy does not help in improving the ALU lifetimes.

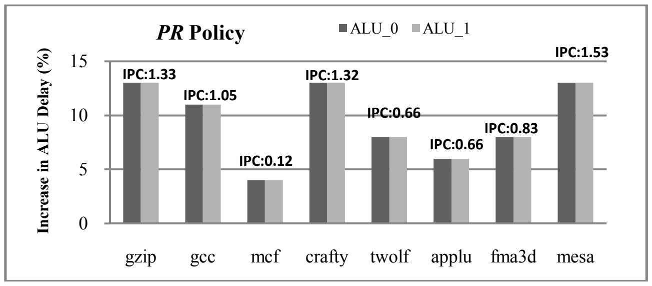

Figure 5: Degradation in ALU lifetime due to the *LB*/*PR* Policy in 2-wide core

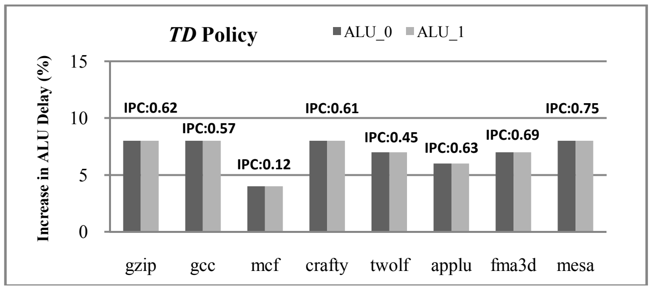

Figure 6: Degradation in ALU lifetime due to the *TD* Policy in 2-wide core

The *TD* policy is again the best in terms of lifetime reliability and ALU_0 achieves a 32% delay improvement although there is an almost equal loss in performance. The higher performance loss as compared to 4-wide issue cores is again due to the higher utilization of both ALUs in the 2-wide core. Benchmarks with higher IPCs gain more reliability improvements at a cost of higher performance drops. Benchmarks with lower IPCs are less sensitive to this policy.

These results show that the *PR* policy is effective for a 4-wide core but not for a 2-wide core. The *TD* policy provides good lifetime reliability but at the cost of performance loss. The key parameter in the *TD* policy is $Cycle_{TD}$. Higher the $Cycle_{TD}$ value, higher the improvement in lifetime, which in turn results in lower performance. In order to study the impact of this parameter, we analyze the *TD* policy with three different $Cycle_{TD}$ values: 1, 2, and 3 cycles. Figure 7 presents the lifetime of ALU_0 for *PS* policy and *TD* policy with three $Cycle_{TD}$ values whereas Figure 8 provides the corresponding performance results in terms of the IPC.

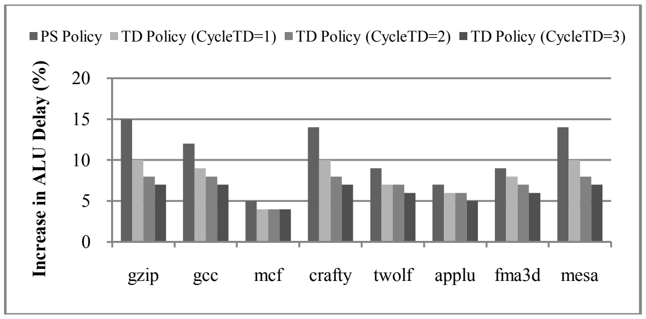

Figure 7: Lifetime impact of the *TD* Policy with $Cycle_{TD}$ = (1, 2, 3) wrt. PS policy on a 2-wide core

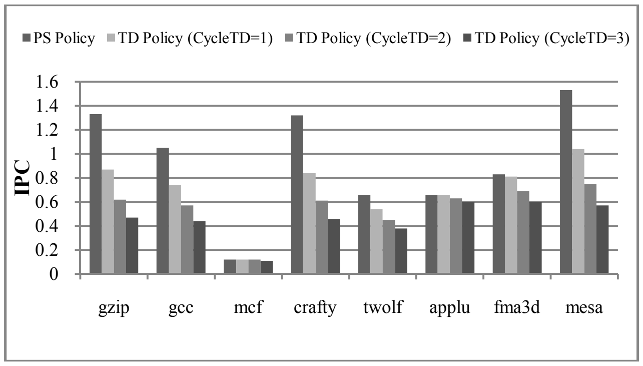

Figure 8: Performance of *TD* Policy with $Cycle_{TD}$ = (1, 2, 3) wrt. PS policy on a 2-wide core

From Figure 7 it is clear that higher $Cycle_{TD}$ provides better lifetime for ALU_0 as it creates more room for recovery. On the other hand, as Figure 8 shows, the performance drops with higher values for $Cycle_{TD}$. On average, a single-cycle of $Cycle_{TD}$ improves lifetime reliability by 23% and decreases performance by 19% with respect to *PS* policy whereas a 2-cycle recovery period shows nearly an equal gain in reliability as loss in performance (approximately 32% each). Finally, a 3-cycle $Cycle_{TD}$ improves lifetime reliability by 39% but causes a 42% drop in performance. We can also observe in Figures 7 and 8 that benchmarks with low IPC are less sensitive to the choice of the *Cycle$_{TD}$* value.

VIII. Implication of the results and future work

The results indicate that there are lifetime reliability and performance tradeoffs between the *PR* and *TD* policies for the two types of cores. *PR* provides good improvements in lifetime reliability for 4-wide issue cores and provides high performance but is less effective for 2-wide issue cores. *TD*, on the other hand, provides good reliability benefits for both types of cores but significantly degrades performance. One can balance between performance and reliability in *TD*-based schedulers by choosing an appropriate *Cycle$_{TD}$* value. These results show that intra-core NBTI management can be viable but requires a careful matching of the scheduling policy to the microarchitecture of the core. These results are especially interesting in the context of reliability-aware heterogeneous multicore processor design. Another implication of these results pertains to whether we would like a core to provide high performance or provide more reliability at any given point of time. Similar to the *Facelift* idea proposed by Tiwari and Torrellas for managing NBTI at the granularity of entire cores [8], one could perform fine-grained reliability management by using the *PR* policy to optimize for performance and switch to the *TD* policy if slowing down aging is the main objective. One could also choose to use different scheduling policies on different cores of a multicore processor based on some optimization criterion. Finally, there have been some recent research efforts on designing circuit-level NBTI sensors [24, 25]. One can use NBTI-aware scheduling techniques in conjunction with such sensors to effectively manage lifetime reliability within each core.

IX. Conclusions

Silicon reliability is one of the key challenges facing the microprocessor industry. Architects have to design processors that are resilient against lifetime reliability. A large body of research exists on tackling lifetime reliability problem, but there has been little work on managing reliability proactively. In this paper, we explore how dynamic instruction scheduling policies can affect lifetime reliability of the resources. We show that the conventional instruction scheduling policy can lead to NBTI problems in the ALUs and propose and evaluate two NBTI-aware policies. We analyze these policies for two different processor core microarchitectures. Our results motivate further research on proactive, intra-core NBTI mitigation techniques.

X. Acknowledgements

We thank the FACT Group at Intel for early feedback on this paper. This research has been supported in part by NSF CAREER Award 0643925, NSF grant 0627527, and gifts from Intel.

References

[1] Mahmut Yilmaz, Albert Meixner, Sule Ozev, Daniel J. Sorin, “Lazy Error Detection for Microprocessor Functional Units,” *IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, September 2007.

[2] Fred A. Bower, Daniel J. Sorin, Sule Ozev, “A Mechanism for Online Diagnosis of Hard Faults in Microprocessor,*” 38th Annual International Symposium on Microarchitecture*, November 2005.

[3] Albert Meixner, Michael E. Bauer, Daniel J. Sorin, “Argus: Low-Cost, Comprehensive Error Detection in Simple Cores,” *40th Annual International Symposium on Microarchitecture*, December 2007.

[4] T. M. Austin, “DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design," *32nd Annual IEEE/ACM International Symposium on Microarchitecture*, November 1999.

[5] Albert Meixner, Daniel J. Sorin, "Error Detection Using Dynamic Dataflow Verification," *International Conference on Parallel Architecture and Compilation Technique*, September 2007.

[6] Jaume Abella, Xavier Vera, Antonio González, “Penelope: The NBTI-Aware Processor”, In *Proc. of the 40th Annual International Symposium on Microarchitecture*, December 2007.

[7] S. Reinhardt and S. Mukherjee, “Transient Fault Detection via Simultaneous Multithreading”, *In Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 25–36, June 2000.

[8] A. Tiwari and J. Torrellas, “Facelift: Hiding and Slowing Down Aging in Multicores”, In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, November 2008.

[9] S. Feng, S. Gupta and S. Mahlke, “Olay: Combat the Signs of Aging with Introspective Reliability Management”, *Workshop on Quality-Aware Design (W-QUAD),*June 2008.

[10] D. Burger and T. Austin. The SimpleScalar Toolset, Version 3.0. http://www.simplescalar.com

[11] D. Brooks, V. Tiwari and M. Martonosi, “Wattch: A Framework for Architectural-Level Power Analysis and Optimizations”, *In Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 83–94, June 2000.

[12] K. Skadron et al. “Temperature-Aware Microarchitecture”, In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 1–13, June 2003.

[13] S. Palacharla, N. Jouppi and J. Smith, “Complexity-effective Superscalar Processors”, *In Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 206-218, June 1997.

[14] SPEC CPU2000. http://www.spec.org/cpu2000/.

[15] T. Sherwood et al, “Automatically Characterizing Large Scale Program Behavior”, *In Proceedings of the International Conference on Architectutral Support for programming languages and Operating Systems (ASPLOS),* October 2002.

[16] A. Phansalkar, A. Joshi, L. Eeckhout, and L. K. John. “Measuring program similarity: Experiments with SPEC CPU benchmark suites”, In *IEEE International Symposium on Performance Analysis of Systems and Software*, March 2005.

[17] W. Wang, V. Reddy and A. Krishnan, “Compact Modeling and Simulation of Circuit Reliability for 65-nm CMOS Technology”, *IEEE Transactions on Device and Materials and Reliability*, 7(4):509-517, December 2007.

[18] J. Shin et al, “A Proactive Wearout Recovery Approach for Exploiting Microarchitectural Redundancy to Extend Cache SRAM Lifetime”, *In Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 83–94, June 2008.

[19] X. Fu, T. Li and J. Fortes, “NBTI Tolerant Microarchitecture Design in the Presence of Process Variation”, In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, November 2008.

[20] J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers, “The case for lifetime reliability-aware microprocessors”. In *Proc. of the 31st Annual International Symposium on Computer Architecture*, June 2004.

[21] J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers, “Exploiting structural duplication for lifetime reliability enhancement”, *In Proc. of the 32nd Annual International Symposium on Computer Architecture*, June 2005.

[22] M.D. Powell, E. Schuchman and T.N.Vijaykumar, “Balancing Resource Utilization to Mitigate Power Density in Processor Pipelines”, In *Proceedings of the 38th Annual International Symposium on Microarchitecture*, November 2005.

[23] T. Sakurai and R. Newton, “Alpha-power law MOSFET model and its applications to CMOS inverter delay and other formulas”, *IEEE Journal of Solid-State Circuits*, April 1990.

[24] Z. Qi, M.R. Stan, “NBTI resilient circuits using adaptive body biasing”, In *Proceedings of the 18th ACM Great Lakes symposium on VLSI*, 2008.

[25] J. Keane, T. Kim, C.H. Kim, “An on-chip NBTI sensor for measuring PMOS threshold voltage degradation” In *Proceedings of the 2007 international symposium on Low power electronics and design*, August 2007.

[26] E. Schuchman and T.N.Vijaykumar, “BlackJack: Hard Error Detection with Redundant Threads on SMT”, In *Proceedings of the 37th International Conference on Dependable Systems and Networks (DSN)*, June 2007.