NBTI Mitigation by Optimized NOP Assignment and Insertion

Farshad Firouzi, Saman Kiamehr, and Mehdi B. Tahoori
Chair of Dependable Nano-Computing (CDNC)
Karlsruhe Institute of Technology (KIT)
76131 Karlsruhe, Germany
Email: {farshad.firouzi, kiamehr, mehdi.tahoori}@kit.edu

***Abstract*—Negative Bias Temperature Instability (NBTI) is a major source of transistor aging in scaled CMOS, resulting in slower devices and shorter lifetime. NBTI is strongly dependent on the input vector. Moreover, a considerable fraction of execution time of an application is spent to execute NOP (No Operation) instructions. Based on these observations, we present a novel NOP assignment to minimize NBTI effect, i.e. maximum NBTI relaxation, on the processors. Our analysis shows that NBTI degradation is more impacted by the source operands rather than instruction opcodes. Given this, we obtain the instruction, along with the operands, with minimal NBTI degradation, to be used as NOP. We also proposed two methods, software-based and hardware-based, to replace the original NOP with this maximum aging reduction NOP. Experimental results based on SPEC2000 applications running on a MIPS processor show that this method can extend the lifetime by 37% in average while the overhead is negligible.**

I. Introduction

As CMOS technology scales to the nano-meter regime, reliability mostly due to transistor aging has become a critical issue of VLSI digital circuits. *Negative Biased Temperature Instability* (NBTI) has a significant effect on the circuit performance and lifetime, and is considered as the dominant concern of reliability [1]. NBTI increases the threshold voltage of PMOS transistors over time and occurs when a negative bias voltage ($V_{gs} = -V_{dd}$) is applied on a PMOS transistor. As a result, this phenomenon leads to degradation of circuit performance over the time. This degradation is strongly dependent on the input patterns and duty cycle (i.e the ratio between the times when the transistor is under negative bias to the total time). Eventually, when the post aging delay of the circuit, exceeds the timing limit, the circuit starts to fail, and the lifetime is considerably reduced.

Over the past few years, several methods have been proposed to alleviate and mitigate the effect of NBTI at various levels of abstraction. Since NBTI strongly depends on the input patterns of the circuit, *Input Vector Control* (IVC) is used in [5], [6] to mitigate the NBTI effect at the circuit level. NBTI-induced delay is tackled in [7] by applying appropriate input vector to the functional units during standby mode (idle time). Unused bits in source operands are exploited in [8] to alleviate NBTI effect on ALU. Authors of [9] proposed a per-buffer-entry based NBTI recovery scheme during idle cycles targeting functional units with storage cells such as RS (reservation stations), ROB (reorder buffers) and PR (physical registers).*Dynamic Voltage and Frequency Scaling* (DVFS), which is traditionally used for reducing power consumption, has been adapted to alleviate the NBTI effect in [3]. Moreover, different NBTI-aware scheduling policies are studied in [10] to address aging effect in functional units.

Due to data and control hazards and memory stalls, pipelined processors need to execute instructions that have no effect on the state of the processor [11]. These special instructions are referred as NOP and their effects are actually to occupy the hardware resources for a certain instruction slots with no effect on program execution. It should be noted that there are multiple cases of instruction which can act as NOP (e.g $SLL\ R0, R0, 0$ or $ADD\ R0, R0, 0$). Since NOPs do not change the state of the executed application, the time spent for executing NOP instructions in a processor can be viewed as a pseudo-idle time. Based on out observation, a considerable fraction of total executed instructions of SPEC2000 benchmark programs are NOP instructions. This implies that, there are plenty of opportunities for alleviating NBTI effect. Indeed, NBTI effect strongly depends on the input vector. Therefore, the impact of the NBTI can be reduced by executing a suitable instruction as a NOP. *The key idea in this paper is finding a new instruction with no effect on the program state to replace the processor's default NOP instruction in order to minimize the NBTI effect.*

A key requirement to successfully exploit a NOP instruction for aging reduction is understanding the effect of different instructions on aging. For this purpose, we investigate the impact of all possible instruction opcodes and instruction source operands on the delay-degradation imposed by NBTI. The results illustrate that the NBTI degradation effects of the instruction opcodes that can be used as NOP are almost the same and minimal. On the other hand, source operands have a significant influence on the amount of NBTI degradation to the processor. Based on this observation, we use a *Linear Programming* (LP) approach for finding the best *Maximum Aging Reduction* (MAR) NOP (opcode and source operand values) which leads to minimum NBTI-induced delay degradation while has no effect on the state of the executed program and acts like a normal NOP. Finally, two different techniques (software-based and hardware-based) are proposed to show how the extracted MAR NOP can be applied to the processor. We evaluate our proposed approach on a MIPS processor with various SPEC2000 benchmark applications in terms of lifetime improvement, power and area overheads. We show that the lifetime of the processor can be extended by 37% in average while the observed area and power overheads are less than 1%.

The rest of the paper is organized as follows. In Section II, we introduce the NBTI model and the micro-architecture of the processor (MIPS) that is used as the case study. The proposed MAR NOP selection methodology is described in Section III. Section IV presents the proposed approaches for implementing the extracted MAR NOP. Experimental results are demonstrated in Section V. Finally, Section VI concludes the paper.

II. Preliminaries

A. NBTI Modeling

NBTI occurs in PMOS transistors when the transistor is under negative gate-source bias i.e., $V_{gs} = -V_{dd}$, (stress mode). When PMOS experiences the stress mode, some Si-H bonds are broken at the interface $Si - SiO_2$ due to the presence of holes in the channel. The resulting H diffuses away and as a result some positive traps (Si^+) are left causing the magnitude of transistor threshold voltage

978-3-9810801-8-6/DATE12/ ©2012 EDAA IEEE

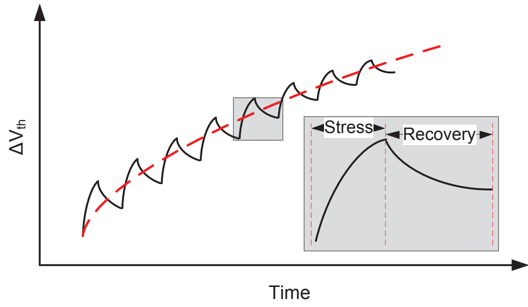

Fig. 1. The conceptual illustration of V_{th} change under stress and recovery conditions

to increase [12]. When the stress mode is removed (recovery mode i.e. $V_{gs} = 0$) some of the created interface traps are healed, and the threshold voltage increase can partially be recovered [12]. Figure 1 shows the NBTI-induced threshold voltage change during stress and recovery cycles.

The long-term NBTI-induced threshold voltage shift, ΔV_{th}, can be modeled by the following equation [13]:

$$\Delta V_{th} = AY^n t^n \tag{1}$$

where A is a technology dependent factor, n is a fabrication process dependent constant, t is the total time, and Y is the duty cycle. The duty cycle represents the ratio between stress to total time. To calculate the delay degradation of a gate due to ΔV_{th} imposed by NBTI, the alpha power model is used [14]:

$$\tau = \frac{K}{(V_{gs} - V_{th})^{\alpha}} \tag{2}$$

Here K is a technology dependent factor, V_{gs} is the gate-source voltage, and α represents the velocity saturation. If ΔV_{th} is small, by using first order Taylor expansion, the following equation for NBTI-induced delay degradation can be derived from Equation (2):

$$\Delta\tau = \frac{\alpha \Delta V_{th}}{V_{gs} - V_{th0}} \times \tau_0 \tag{3}$$

where ΔV_{th} is the NBTI-induced threshold voltage change, V_{th0} is the original transistor threshold voltage, and τ_0 is the pre-aging delay of the gate. By using Equations (1) and (3), the total NBTI-induced delay degradation can be estimated as follows:

$$\Delta\tau = BY^n t^n \tau_0 \tag{4}$$

where

$$B = \frac{A\alpha}{(V_{gs} - V_{th})^{\alpha}}$$

It should be noted that, the above equation is valid for simple inverter which has one transistor in each pull-up/pull-down network. For other gates (e.g. NAND and NOR) the stacking effect has to be considered as well. In stacked structures, the delay of each gate is affected by the threshold voltage change of all the internal transistors. Moreover, the duty cycle of each transistor depends not only on the state of its input, but also on the state of upper and lower transistors [15].

B. The MIPS Architecture

In this section, we briefly overview the MIPS (Microprocessor without Interlocked Pipeline Stages) in-order pipelined processor. Please note that our methodology is general and can be easily applied to other embedded processors. The choice of MIPS is due to its simplicity for better explanation of our method, and its widespread use in academia and industry. Moreover most modern embedded processors, superscalars, and multi-core systems are based on the MIPS multistage pipeline structure [16], [17]. The classic MIPS pipeline consists of the following five stages: 1.Instruction fetch (IF), 2.Instruction decode and register file read (ID), 3.Execution or address calculation (EX), 4.Data memory access (MEM), 5.Write-back (WB).

There are situation called *Hazards*, in which the next instruction in the instruction sequence stream cannot be executed during its designated clock cycle and it is resolved by inserting NOP instructions. In addition to *data hazards*, which are caused by operand dependencies and can often be resolved by data forwarding, and *control hazards* (branches), there are *memory stalls* (due to memory access and cache misses) which are all resolved by flushing the pipeline and/or inserting NOPs in the pipeline.

III. MAR NOP SELECTION

In this section, we describe our proposed technique for reducing the delay degradation imposed by NBTI based on modifying the NOP instruction. The NBTI-induced degradation which each transistor experiences strongly depends on the state of its input. Therefore, input vector considerably affects the NBTI-induced delay degradation of the circuit. Besides, NOPs are special instructions which are inserted during the application execution to solve hazards, memory stalls, and dependency problem. Although these instructions are executed and use the hardware resources, they do not change the state of the executed program. Moreover, a considerable portion of the execution time is spent executing NOPs. Putting all together, the key idea of this paper is finding a suitable NOP instruction which maximizes recovery during the NOP execution and hence minimize the NBTI effect on a processor. First, we investigate the NBTI effect of all possible instructions which can potentially be used as a NOP instruction. Next, we introduce a Linear Programming (LP) approach for selecting maximum NBTI reduction NOP.

A. NBTI effect of Possible NOPs

NOP is an instruction with no effect on the program execution and since it has a neutral effect, it can be inserted at any location in the program execution. For example in a MIPS processor the default NOP instruction is:

$$SLL\ R_0,\ R_0,\ 0$$

This instruction denotes, the content of R_0 is shifted left zero times. Since the R_0 is hardwired to 0, this instruction has no effect on the status of the program. Many instructions such as ADD, OR, SUB with R_0 or immediate operands can be used alternatively as a NOP. It should be noted that, using any other register rather than R_0 even as a source operand, may cause a data hazard. For example, the following instruction is an alternative for default NOP.

$$ADDI\ R_0,\ R_0,\ 8$$

Both introduced NOPs (default NOP and the ADDI example) can be used as NOP with no effect on program execution. However, since they have different opcode and source operands, they may cause different amount of NBTI-induced delay degradation on the processor. We investigate the effect of applying different NOP candidates on NBTI-induced delay degradation of the processor, based on the flowchart depicted in Figure 2(a). First, a logic synthesis tool is used to map the processor to a gate-level description. Besides, all critical paths of the processor are extracted using a Static Timing Analysis

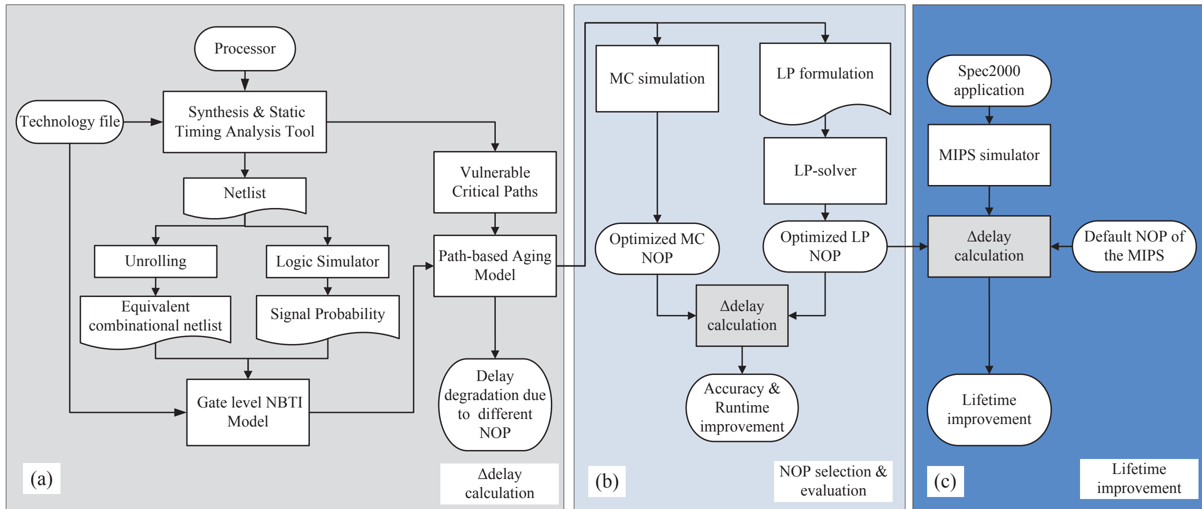

Fig. 2. Overall flow of the proposed NBTI-aware NOP selection and evaluation

(STA) tool. The netlist extracted from the logic synthesis tool is given to a logic simulator to obtain the signal probabilities (SP) of internal nodes. SP of a node is the probability of being one and represents the duty cycle of the corresponding transistors. Since processor is a sequential circuit, unlike combinational circuits, it contains some flip-flop (for example pipeline registers) and feedbacks. For example in MIPS, some inputs of the ALU depend on the output of the forwarding unit which themselves are the outputs of the ALU in a previous cycle. As a result, unrolling has to be performed n-times to remove feedbacks and flip-flops. Where n is the sequential depth of the circuit which here is equal to the number of pipeline stages. The output of the unrolling process (a pure combinational logic) as well as SPs are inputs of the gate-level NBTI model introduced in Section II-A. Finally, the NBTI-induced delay degradation is calculated considering all critical paths.

Based on the introduced methodology, we investigate the effect of different instruction opcodes and operands on the NBTI-induced delay degradation of the processor. For each instruction opcode, the delay degradation imposed by NBTI is calculated for 100,000 randomly generated source operands (e.g. the immediate values and the data stored in source registers). According to the results illustrated in Figure 3, the average of the delay degradations for all instruction opcodes are almost the same. On the other hand, the variation of the NBTI-induced delay degradation is very sensitive to the values of source operands (the ranges shown in Figure 3). Therefore, it can be concluded that NBTI-induced delay degradation of a processor is mostly affected by the source operands rather than the instruction opcode. This phenomenon is mainly due to the following reasons:

- According to the analysis of critical paths, most of the aging vulnerable paths which can change the post-aging delay of the processor, are located in the EX-stage. Since the instruction opcode mostly affects the decoder rather than the EX-stage (specially ALU), the effect of the instruction opcode on the NBTI-induced delay degradation of the processor is negligible.
- Since the width of the opcode is considerably smaller than the operand width, the number of gates affected by the opcode is less than those affected by source operands.

Based on the above observations, to exploit NOP as a mechanism to efficiently minimize NBTI effect, one need to choose both opcode instruction and the values of source operands of NOP precisely. As a result two concerns should be addressed. First, optimized source

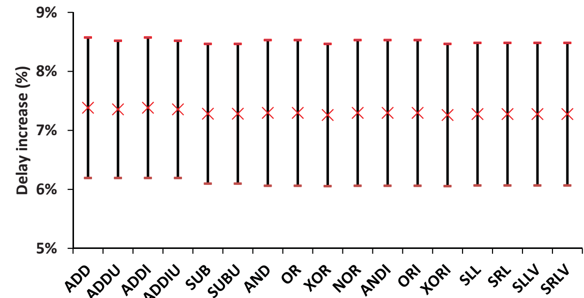

Fig. 3. The effect of different NOPs (opcode and operand values) on NBTI-induced delay degradation (the range shows the impact of operand values)

operand values should be obtained for each opcode instruction in terms of NBTI. For this purpose, a Linear Programming approach is presented. Second, a mechanism should be devised to replace the default NOP and apply the opcode and its corresponding optimized source operands as a MAR NOP.

B. Linear Programming Approach

The straightforward solution for finding MAR NOP is to exhaustively apply and analyze all possible NOPs (and operand values) which is infeasible. In this section we present a Linear Programming (LP) approach for obtaining MAR NOP which results in minimum NBTI-induced delay degradation. The main characteristic of NOPs is that they do not change the state of the program. Therefore, only a subset of the instruction set can be employed as NOP. Moreover, as observed in Section III-A, the NBTI effect mostly depends on the source operand rather than the instruction opcode. As a result, we need to modify NOP in a way that, it consists of not only a suitable instruction opcode, but also the corresponding NBTI-optimized operands value. To find a MAR NOP, first, all possible instruction opcodes that can be act as a NOP are considered. The first column of Table III shows all possible instruction opcodes that can be used as NOP instruction opcode. Next, for each opcode, the best operand values which minimize the NBTI effect on the processor is extracted. Due to the large input set of processor (number of operand values), we exploit an LP approach to find the optimized operand value for each opcode. Finally, the pair of opcode and corresponding

operand values resulting in the minimum delay degradation is selected as MAR NOP.

For each opcode, the objective is to minimize the overall post aging delay of the processor imposed by NBTI considering all critical paths. The result of this optimization is a source operand leading to minimum NBTI of a processor for each instruction opcode. This objective can be represented by the following equation:

$$minimize : x \mid \forall j : \; x \geq \tau(CP_i) = \sum_{g_i \; in \; CP_j}^{i} \Big(\tau(g_i) + \Delta\tau(g_i)\Big) \quad (5)$$

where CP is a critical path and g_i is the gate i in the critical path. For each critical path, the post-aging delay is the sum of the post-aging delay of all the gates along that path. The post-aging delay of each gate is equal to summation of pre-aging delay of the gate ,$\tau(g_i)$, and the NBTI-induced delay increase ,$\Delta\tau(g_i)$,. As mentioned before, the NBTI-induced delay increase of each gate depends on the state of its inputs. To represent $\Delta\tau(g_i)$ of each primitive gate in a LP compatible format we use the equations shown in Table I [5].

TABLE I
LP COMPATIBLE OBJECT FUNCTIONS FOR GATE ΔDELAYS

Function	Logic operation	Object function
INV	$z = NOT(x)$	$\tau_0 z + \tau_1 x$
NAND	$z = NAND(x, y)$	$(\tau_{10} - \tau_{00})x \; + \; (\tau_{01} - \tau_{00})y \; +$ $(\tau_{10} + \tau_{01} - \tau_{00} - \tau_{11})z$ $+(2\tau_{00} + \tau_{11} - \tau_{01} - \tau_{10})$
NOR	$z = NOR(x, y)$	$(\tau_{11} - \tau_{01})x \; + \; (\tau_{11} - \tau_{10})y \; +$ $(\tau_{11} + \tau_{00} - \tau_{10} - \tau_{01})z$ $+(\tau_{10} + \tau_{01} - \tau_{11})$

where ($x \; and \; y \in \{0, 1\}$) are the inputs of the gate, z is the output of the gate, and τ_{xy} indicates the NBTI-induced delay increase corresponding to the gate inputs xy. Moreover, the functionality of the circuit is required to be represented as a set of linear constraints. Table II shows the LP compatible constraints for basic logic gates.

TABLE II
LP CONSTRAINTS FOR BASIC LOGIC OPERATIONS

Function	Logic operation	LP constraints
INV	$z = NOT(x)$	$z + x = 1$
NAND	$z = NAND(x, y)$	$z \leq 2 - x - y$ $z \geq 1 - x$ $z \geq 1 - y$
NOR	$z = NOR(x, y)$	$z \geq 1 - (x + y)$ $z \leq 1 - y$ $z \leq 1 - x$

IV. APPLYING MAR NOPS

In this section, we present two different methods for applying the instruction opcodes and their corresponding optimized source operand values as an MAR NOP.

A. Software-based implementation

In order to apply the operand values of MAR NOP without affecting the program execution, we need to reserve some registers. These registers are dedicated only to save the corresponding operands of the MAR NOP (not available for application anymore) and are loaded right before the application is executed. Table III shows all possible instructions of MIPS which can be used as a NOP instruction. For example, $ADD \; R0, R_i, R_j$ can act as a NOP instruction only if the registers R_i and R_j are reserved. Otherwise, since the application might use these registers, the output of the program could be affected.

TABLE III
NOP CANDIDATES OF MIPS PROCESSOR IN THE SOFTWARE-BASED IMPLEMENTATION

Operation (OP)	Operand	# of reserved registers
ADD, ADDU, SUB SUBU, XOR	$R_0 \leftarrow R_i \; OP \; R_j$	2
	$R_i \leftarrow R_i \; OP \; R_0$	1
OR	$R_0 \leftarrow R_i \; OP \; R_j$	2
	$R_i \leftarrow R_i \; OP \; R_0$	1
	$R_i \leftarrow R_i \; OP \; R_i$	1
ADDI, ADDIU ORI, XORI	$R_0 \leftarrow R_i \; OP \; Imm$	1
	$R_i \leftarrow R_i \; OP \; 0$	1
AND	$R_0 \leftarrow R_i \; OP \; R_j$	2
	$R_i \leftarrow R_i \; OP \; R_i$	1
ANDI	$R_0 \leftarrow R_i \; OP \; Imm$	1
	$R_i \leftarrow R_i \; OP \; 1$	1
NOR	$R_0 \leftarrow R_i \; OP \; R_j$	2
SRA, SLL, SRL	$R_0 \leftarrow R_i \; OP \; SA$	1
	$R_i \leftarrow R_i \; OP \; 0$	1
SRLV, ROTRV SLLV,SRAV	$R_0 \leftarrow R_i \; OP \; R_j$	2
	$R_i \leftarrow R_i \; OP \; R_0$	1
ROTR	$R_0 \leftarrow R_i \; OP \; SA$	1
	$R_i \leftarrow R_i \; OP \; 0$	1
	$R_i \leftarrow R_i \; OP \; 32$	1
Default NOP of MIPS	$R_0 \leftarrow R_0 \; SLL \; 0$	0

The last column of Table III shows the number of registers needed to be reserved for the corresponding NOP instructions. It should be noted that, these instructions are selected in a way that, the data stored in the reserved registers does not change during the NOP execution. In other words, the reserved registers keep the NBTI-optimized source operand during NOP execution. In conclusion, to apply a MAR NOP in software-based approach the following steps are performed:

1) Modify the compiler directives to generate the binary/assembly code while reserving the required (one or two) registers
2) Replace the default NOPs in the code with MAR NOP
3) Add necessary instructions to the beginning of the code to assign those reserved registers to the optimal values of MAR operands

Another alternative of software realization of MAR NOP is round-robin allocation of the registers to the operands of the MAR NOP, however, it requires further modifications to the compiler.

B. Hardware-based implementation

Here we present a hardware-based method for replacing the default NOP with the MAR NOP applying them during program execution. In this approach, we modify the input multiplexers of the ALU in a way that the NBTI-optimized source operand for the NOP-instruction is directly provided in the EX-unit (see Figure 4). For this purpose, an extra input is added to each of the input multiplexer of the ALU. These inputs provide the NBTI-optimized data for MAR NOP. In addition, decoder should be slightly changed accordingly to support the modification of the input multiplexer of the ALU. Since the operand values of the NOP are available in the EX-stage, the hardware-based NOP implementation can handle all the situations stem in hazards (e.g when a NOP is needed to be inserted from the EX-stage into the processor). Moreover, to insert a NOP instruction from IF stage, due to branch hazard, the Hazard Detection Unit should

be accordingly changed to reset the instruction field of the IF stage to the MAR NOP (In base MIPS processor, this field is set to the default NOP).

C. Comparing Hardware vs Software Implementations

As mentioned, the software-based NOP implementation needs at least one reserved register. This implies that, the number of available registers for compiling a program is reduced. As a result, the performance might be decreased. Another limitation of the software-based approach is that, it cannot be used for all types of hazard which have been handled by the traditional NOP. There are some situations which might occur when the forwarding unit cannot avoid data hazards. As an example, consider an instruction which needs data that is provided by a preceding load instruction. In this case, the Hazard Detection Unit in the ID-stage identifies this situation in advance and inserts a stall between these dependent instructions. As a result, we should force the EX stage to perform a special instruction that does not change the state of the processor. Since here NOP is inserted from EX stage, software-based method cannot handle this situation. This is due to the fact that the source registers are read in the ID-stage and since this type of NOP is inserted after the ID-stage, the registers which contain the NBTI-optimized operands cannot be read. This phenomenon might also occur in control hazards. The most well-known method for resolving control hazards and reducing the branch penalty is branch prediction method. In case of misprediction, all of the instructions fetched according to the prediction, are flushed from the pipeline. To flush an instruction, the instruction field of the register is set to a NOP instruction. Depending on the branch execution unit, NOPs might be inserted from the EX-stage. Similar to the previous situation in case of data hazards, software-based MAR NOP implementation cannot be used here as well. In order to overcome these cases, compiler can be configured to take care of all possible hazards and stalls and hence bypass the hardware supports by applying the necessary NOPs statically at the compile time. However, this may result in some performance degradation, particularly for branches.

Despite the discussed limitations of software-based implementation, this approach is flexible and does not need any special hardware support or modification. In addition, since the critical paths of the circuit might change due to the different data patterns of the applications during the circuit lifetime, the optimized operands of the MAR NOP might change as well. Changing the appropriate operands of the MAR NOP in a software-based implementation is very straightforward and only needs to load new operands into the corresponding reserved registers before the program execution.

The main advantage of the hardware-based implementation of MAR NOP is that it can be used for all situations when a NOP must be inserted to the processor even from the EX-stage. However, since the optimized operands of the MAR NOP are provided by a hardwired method, it is not as flexible as the software-based implementation. To remove this drawback it is possible to use the already available scan-chain registers for modifying the operands according to the current state of the processor in terms of timing properties of the critical paths, with some additional design changes and overhead.

In in-order processors, when an instruction cannot be executed due to hazards and stalls, all the following instructions should be stalled and NOP instructions are inserted until the new instruction can be executed. On the other hand, in out-of-order superscalar processors, Functional Units (FUs), such as ALU, reservation stations, reorder buffers and physical registers, are isolated from each other with some sort of buffers. Therefore, when an instruction faces a hazard or stall, the following instructions can be executed. Typically, the utilization of FUs is far less than 100% and clock-gating is used during idle cycles to reduce power consumption. Therefore, the idle time of each FU can be exploited to apply MAR NOP. For each FU, based on its functionality and gate-level implementation, a suitable MAR NOP can be extracted and can be applied to the corresponding FU for any cycle that the FU is idle. In this case, the hardware-based implementation of MAR NOP is more favorable because software-based implementation might lead to some performance overhead.

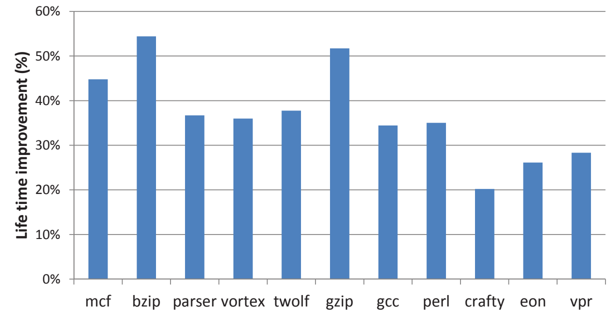

Fig. 5. Lifetime improvement for selected spec2000 application using NBTI-aware NOP assignment

V. EXPERIMENTAL RESULTS

To evaluate the efficiency of the proposed method a five-stage MIPS processor is used. It should be noted that our methodology is generic and can be applied to other processors. The processor is synthesized by Synopsys Design Compiler and is mapped to TSMC 65nm standard cell library. By assuming a delay degradation of 10% in 3 years, all critical paths with 10% positive slack are extracted using PrimeTime static timing analyzer. Then, the unrolling method is applied to remove the logic feedbacks and convert the sequential structure of the processor to a combinational one. The signal probability (duty cycle) of each transistor is calculated by a logic simulator as well (see Figure 2(a)).

By the method presented in III-B the problem of finding the best source operand for each opcode instruction is obtained using CPLEX (LP solver) and then the best pair (opcode and corresponding operand values) is selected as MAR NOP. We have also implemented a Monte Carlo (MC) simulation to obtain the optimal source operand for each opcode instruction to validate the LP approach. The MC approach is terminated when the obtained solution is not improved more than 0.1% after 100,000 iteration (see Figure 2(b)). According to the results, due to large input set of the processor, the MAR NOP obtained by LP is better optimized than MC (5%) while reduces the runtime by 150x in average.

To analyze the efficiency of the extracted MAR NOP on the processor lifetime, we choose SPEC2000 benchmarks. We do a profiling on these applications with the M5 simulator [18]. Based on the output of the profiling, extracted MAR NOP, and default MIPS NOP the lifetime improvement is calculated by using the flow depicted in Figure 2(c). Figure 5 shows the lifetime improvements for the SPEC2000 applications, when the default NOPs are replaced with MAR NOPs. According to the results our proposed approach can extend the lifetime of the processor by 37% in average. It should be noted that the actual results strongly depend on the technology node, circuit design, and architecture which varies from one processor to another.

The software-based implementation needs to reserve one or two registers for storing the optimized source operands of the MAR NOP. To investigate the effect of register reservation on the performance

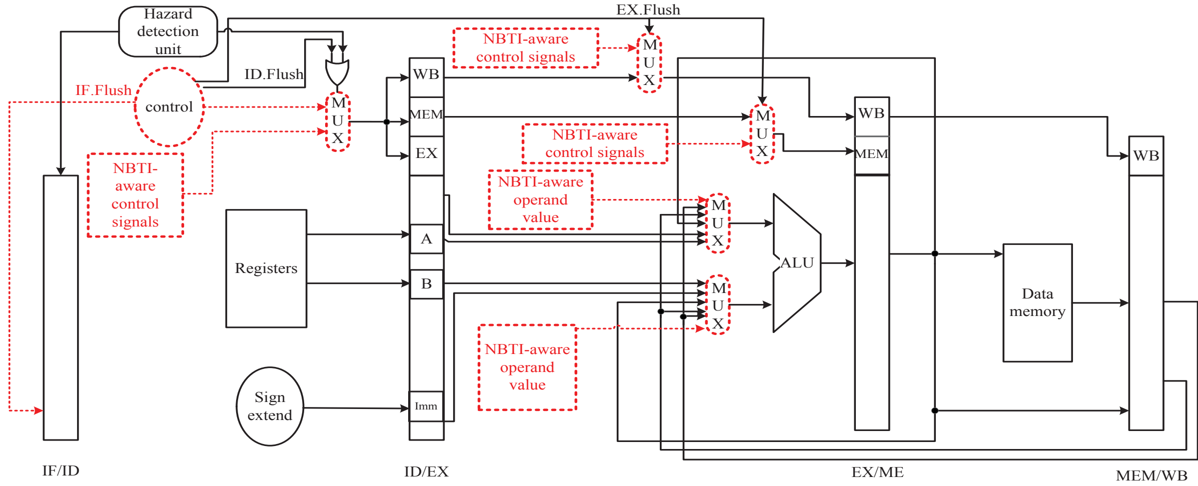

Fig. 4. Hardware-based implementation of NOP in MIPS architecture

of the processor, we apply it to several selected SPEC2000 benchmarks. Each of the application is compiled with gcc-3.4.3 with -O1. According to the results illustrated in Table IV, reserving one register registers reduce the Instructions Per Clock (IPC) by only 0.1%. Moreover, IPC decreased around 0.5% due to reservation of two registers.

TABLE IV
REGISTER RESERVATION OVERHEAD ON IPC

Application	One register	Two registers
mcf	0.0%	0.2%
bzip2	0.0%	0.0%
parser	0.1%	0.5%
vortex	0.0%	-0.4%
twolf	0.0%	0.4%
gzip	0.0%	2.1%
gcc	0.6%	1.1%
perl	0.0%	0.0%
Average	0.1%	0.5%

To analyze the hardware-based approach, the modifications, according to Section IV, have been applied to the RTL description of the MIPS processor and the modified version is synthesized with Synopsys Design Compiler. The results, as shown in Table V, confirm that the overhead of this approach is quite negligible.

TABLE V
NORMALIZED OVERHEAD OF HARDWARE-BASED IMPLEMENTATION OF NOP TO ORIGINAL MIPS

	Original	Modified	Overhead
$Power(mW)$	1.897	1.919	1.1%
$Area(\mu m^2)$	35591	35717	0.3%
$Delay(ns)$	4.38	4.38	0.0%

VI. CONCLUSIONS

As CMOS technology enters advanced nanometer regime, transistor aging mostly due to Negative Bias Temperature Instability (NBTI) is becoming a major reliability concern. NBTI has a strong dependency on the input vector of the circuit. Since NOP (No Operation) instruction is a considerable contributor of total execution time in processors, it can be exploit to tackle the NBTI effect. In this paper, we replace the default NOP with a neutral instruction resulting in maximum aging reduction. We have presented an effective flow to obtain such instructions. We proposed both software-based and hardware-based approaches to apply such MAR NOPs. The results show that this method can extend lifetime by 37% in average, with negligible performance, power, and area overhead.

REFERENCES

[1] K. Bernstein, D. J. Frank, A. E. Gattiker, W. Haensch, B. L. Ji, S. R. Nassif, E. J. Nowak, D. J. Pearson, and N. J. Rohrer. High-performance CMOS variability in the 65-nm regime and beyond. *IBM Journal of Research and Development - Advanced silicon technology*, 50:433–449, 2006.

[2] Y. Wang, X. Chen, W. Wang, Y. Cao, Y. Xie, and H. Yang. Leakage power and circuit aging cooptimization by gate replacement techniques. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, (99):1–14, 2011.

[3] A. Tiwari and J. Torrellas. Facelift: Hiding and slowing down aging in multicores. In *International Symposium on Microarchitecture*, pages 129–140, 2008.

[4] J. Srinivasan, S.V. Adve, P. Bose, and J.A. Rivers. Lifetime reliability: Toward an architectural solution. *Micro, IEEE*, 25(3):70–80, 2005.

[5] F. Firouzi, S. Kiamehr, and M.B. Tahoori. A linear programming approach for minimum nbti vector selection. In *Great lakes symposium on Great lakes symposium on VLSI*, pages 253–258. 2011.

[6] Y. Wang, X. Chen, W. Wang, V. Balakrishnan, Y. Cao, Y. Xie, and H. Yang. On the efficacy of input Vector Control to mitigate NBTI effects and leakage power. In *Quality Electronic Design Int'l Symp.*, pages 19–26, 2009.

[7] J. Abella, X. Vera, et al. Penelope: The nbti-aware processor. In *micro*, pages 85–96. IEEE Computer Society, 2007.

[8] X. Fu, T. Li, and J. Fortes. Nbti tolerant microarchitecture design in the presence of process variation. In *International Symposium on Microarchitecture.*, pages 399–410, 2008.

[9] L. Li, Y. Zhang, J. Yang, and J. Zhao. Proactive nbti mitigation for busy functional units in out-of-order microprocessors. In *Design, Automation and Test in Europe*, pages 411–416, 2010.

[10] T. Siddiqua and S. Gurumurthi. A multi-level approach to reduce the impact of nbti on processor functional units. In *Great lakes symposium on VLSI*, pages 67–72, 2010.

[11] J.L. Hennessy, D.A. Patterson, and D. Goldberg. *Computer architecture: a quantitative approach*. Morgan Kaufmann, 2003.

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

[13] W. Wang, S. Yang, S. Bhardwaj, S. Vrudhula, F. Liu, and Y. Cao. The impact of NBTI effect on combinational circuit: modeling, simulation, and analysis. *Very Large Scale Integration (VLSI) Systems, IEEE Trans.*, 18(2):173–183, 2010.

[14] K.A. Bowman, B.L. Austin, J.C. Eble, X. Tang, and J.D. Meindl. A physical alpha-power law MOSFET model. In *Low power electronics and design*, pages 218–222, 1999.

[15] K.C. Wu and D. Marculescu. Joint logic restructuring and pin reordering against nbti-induced performance degradation. In *Design, Automation and Test in Europe*, pages 75–80, 2009.

[16] *Sun Microsystems. OpenSPARC T1 Microarchitecture Specifications*. 2006.

[17] H.Q. Le, W.J. Starke, J.S. Fields, F.P. O'Connell, D.Q. Nguyen, B.J. Ronchetti, W.M. Sauer, E.M. Schwarz, and M.T. Vaden. Ibm power6 microarchitecture. *IBM Journal of Research and Development*, 51(6):639–662, 2007.

[18] N.L. Binkert, R.G. Dreslinski, L.R. Hsu, K.T. Lim, A.G. Saidi, and S.K. Reinhardt. The m5 simulator: Modeling networked systems. *Micro, IEEE*, 26(4):52–60, 2006.