Wearout-Aware Compiler-Directed Register Assignment for Embedded Systems

Fahad Ahmed[1], Mohamed M. Sabry[2], David Atienza[2] and Linda Milor[1]
[1]Georgia Institute of Technology, Atlanta, GA, 30332, USA
[2]Ecole Polytechnique Federale de Lausanne, 1015 Lausanne, Switzerland
Email: [fahad.ahmed, linda.milor]@ece.gatech.edu and [mohamed.sabry, david.atienza]@epfl.ch

Abstract—**Although constant technology scaling has resulted in considerable benefits, smaller device dimensions, higher operating temperatures and electric fields have also contributed to faster device aging due to wearout. Not only does this result in the shortening of processor lifetimes, it leads to faster wearout resultant performance degradation with operating time. Instead of taking a reactive approach towards reliability awareness, we propose a pre-emptive route toward wearout mitigation. Given the significant thermal and stress variation across the components of microprocessors, in this work we focus on one of the most likely candidates for overheating and hence reliability failures, the register file. We propose different wearout-aware compiler-directed register assignment techniques that distribute the stress induced wearout throughout the register file, with the aim of improving the lifetime of the register file, with negligible performance overhead. We compare our results with a state-of-the-art thermal-aware compilation scheme to show the clear advantage our proposed wearout-aware scheme has over thermal-aware schemes in terms of lifetime improvement that can reach up to 20% for Bias Temperature Instability.**

Keywords: wearout, compiler, register file, lifetime degradation, device aging.

I. INTRODUCTION

Shrinking feature sizes, the inability of the operating voltage to scale accordingly with device dimensions and the ever increasing device density have resulted in decreasing device lifetime [1]. Post-silicon reliability is still heavily dependent on reactive measures, such as error detection and error correction. The use of on-chip sensors [2], complemented by post-silicon tuning [3], is also steadily increasing with the aim of extending system lifetime. However, since they do not mitigate the aging process itself, the effectiveness of such measures is always constrained due to the following reasons. Firstly, in the absence of redundancy, they do nothing in case of catastrophic events such as dielectric breakdown. Secondly, if the amount of aging can be reduced during the lifetime of a device, as proposed in this work, then any post-silicon tuning performed to compensate for wearout can also be regarded as *over design*, since the amount of tuning needed could potentially have been reduced. After all, tuning of a parameter always comes at the cost of reduced system performance.

Modern compilers provide various optimization options for improving performance. However, the flexibility afforded by such compilers is rarely used towards reliability improvement. There has been some focus on compiler-directed thermal-aware register file assignments [4], since their relatively small area and high utilization make them one of the most likely candidates for overheating [5]. Since all wearout mechanisms have a strong dependence on temperature, this inherently leads to improvement in register file reliability. However, to the best of our knowledge, adaptive compilation schemes today are not truly wearout-aware with a primary focus on the extension of the register file's operable lifetime.

The work presented here proposes a wearout-aware compiler directed register assignment scheme that distributes the electrical stress throughout the register file, at compilation time, with the aim of minimizing wearout. The proposed compilation flow consists of two phases of optimization. During phase 1 of the flow, crude wearout estimates are used to perform reliability optimization during the compilation process, while phase 2, which is more time intensive, performs the optimization using detailed wearout models. To support register-window based architectures, this is done at two levels i.e. at the register window level using an adaptive register window assignment, called variable multi-window context switching (VMWCS), and at the register level using an adaptive intra-window register assignment scheme, called adaptive variable assignment (AVA). AVA distributes the stress induced wearout for the registers throughout the window more uniformly with the aim of eliminating registers which might fail fairly early in their projected lifetime. Similarly, adaptive register-window assignment aims at distributing the workload more uniformly at the register-window level. Moreover, as temperature is so critical for device reliability, the proposed wearout-aware register assignment also reduces the temperature of the registers or the register windows.

We also present a framework for extending device level wearout models to the architecture level, which is then used for comparison between different compilation techniques. To do this, our algorithms have been integrated into the CoSy compiler development system [6], a flexible and easily-targetable development kit for high-end compiler designs. Although this work focuses on the register file, we demonstrate the incorporation of wearout in architecture simulations. Hence, the methodology proposed can be extended to other blocks.

The rest of the paper is organized as follows. Section 2 gives a brief overview of the related work and recent trends. In Section 3 we present our pre-silicon wearout prediction framework. Then in Section 4, we present the proposed wearout-aware compilation technique. Next, in Section 5, we present a test case and compare the reliability figures attained

978-1-4673-1036-9/12/$31.00 ©2012 IEEE

with the proposed scheme. Section 6 summarizes the main conclusions of this work.

II. Related Work

Pre-silicon wearout-aware design steps ranging from device level wearout models [7],[8] to wearout augmented CAD tools [9],[10], have become a necessity. The aim of these steps is the prediction of device wearout during its functional lifetime, which can then be complemented with measures, such as the introduction of guard bands, to insure reliable system operation [11]. Post-silicon techniques such as power gating [12] and insertion of control circuit blocks [13] have also been used to reduce wearout.

Recent work in reliability aware compilation has dealt with improving the thermal profile of the functional units under study. Some researchers have used an indirect approach by improving operating temperatures through improvement in power consumption [14]. Moreover, recent work [10] has shown that better thermal profiles can be achieved by considering global temperatures as the main optimization metric. However, an optimized thermal profile does not necessarily result in an optimized lifetime, since device wearout is not only a function of temperature, but also of stress.

Wearout aware scheduling techniques for stress distribution have been used for extending the lifetime of multi-core systems [15]. While this approach requires an additional central management unit which keeps track of and responds to the gradual system wearout, the use of an under-used computation unit as a system manager has also been proposed [16]. An alternative to stress distribution is to gracefully drop failing components from operation, given available redundancy [17].

However, in this work we show that the *intra-component compiler-level* stress distribution, coupled with improved operating temperatures, can lead to an optimized component wearout profile. With improvement in component lifetimes as the main metric, we simultaneously reduce component temperatures and achieve a uniform stress distribution for optimal reliability.

III. Pre-silicon Wearout Prediction

The first step in insuring reliable system operation is to bridge the gap between the established device level wearout models and system behavior at the architecture level. The current mean time to failure (MTTF) based high level reliability models, such as [10], only provide us with crude, single point, reliability estimates based on the assumption that the system is a series failure system. Due to the inherent lack of depth of these models, it becomes impossible to address the core issues affecting system reliability at the architecture level. Hence, in this section we present a detailed wearout model for the processor micro-architectural components.

Figure 1 summarizes the framework employed for estimating the pre-silicon wearout profile of the register file. The HW/SW emulation platform, similar to the one presented in [18], was used to extract the required thermal profile and block level switching activity for a chip running benchmark software. The activity at the input of the block under study (the register file in this case) was converted to stress probabilities and transition probabilities at the inputs. These probabilities were then propagated within the block, depending on its logic behavior, for a complete stress/transition probability profile of the internal nodes of the block under study, as shown in Figure 1. Thus we have the probability of a transition occurring at any node, which can then be translated into the flow of current through the interconnects connected to that node. Similarly, the probability of stress at a node within the block is translated into the probability of voltage stress for the transistors connected to that node. Then, using the thermal profile from the HW/SW emulation platform and the calculated probability of current flow and voltage stress, we can use device level models to characterize any wearout mechanism in the block under study to determine the wearout profile of the system.

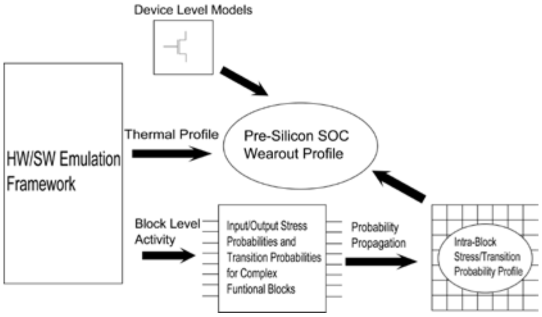

Device Level Models
HW/SW Emulation Framework
Thermal Profile
Pre-Silicon SOC Wearout Profile
Block Level Activity
Input/Output Stress Probabilities and Transition Probabilities for Complex Funtional Blocks
Probability Propagation
Intra-Block Stress/Transition Probability Profile

Figure 1: General framework for estimating the pre-silicon SoC wearout profile.

Finally, we rely on established device level models for the modeling of different wearout mechanisms [7],[8],[19],[20],[21]. Appropriate modifications were made in the models to take into account the probabilistic nature of the input signals. In the next subsection we provide a brief overview of the device level models used in this work.

A. Device Level Modeling for Wearout

Almost all wearout mechanisms can be divided into two broad categories, the voltage (or E-field) dependent wearout mechanisms, such as negative bias temperature instability (NBTI), positive bias temperature instability (PBTI), transistor and backend dielectric breakdown, etc., and the current stress-dependent wearout mechanisms, such as electromigration (EM). Due to the lack of higher level models for the progressive effect of these mechanisms, it was necessary to first model their effect at the device level and then abstract the models to the micro-architecture level. For this work we have focused on three critical wearout mechanisms, namely NBTI, PBTI and transistor gate oxide breakdown (GOBD). Support for the type of dielectric was also included in the modeling framework with poly+SiON and high-k+metal being the available options.

1) Bias Temperature Instability (BTI)

Bias Temperature instability (BTI), as the name suggests, causes instability in device behavior and is a result of the bias stress applied to it. NBTI is the degradation of a PMOS device under negative stress, and PBTI is the degradation of an NMOS device under positive stress. BTI results in shifts in device parameters, such as threshold voltage, transconductance, device mobility, etc., but is generally identified by shifts in the threshold voltage [7].

Historically, BTI was only a major concern for PMOS devices with NMOS devices showing comparatively negligible degradation. However with the introduction of high-k metal gate stacks for sub-45nm technology nodes, degradation in NMOS devices due to positive bias has increased with large degradation observed for both types of devices [22].

Based on the well-known reaction-diffusion (RD) theory, NBTI is directly related to the formation of interface traps and their subsequent diffusion into the dielectric. The corresponding increase in threshold voltage (V_{tp}) follows a power law model[7],[23]:

$$\Delta V_{tp} = A_0 \exp\left(-E_a / kT\right) V^{\alpha} t^{n} \qquad (1)$$

where E_a is the activation energy, *k* is Boltzmann's constant, *T* is temperature, *V* is the voltage stress, and *t* is time. A_0, α, and *n* are fitting parameters and were obtained from experimental results[19].

Upon the release of the voltage stress, there is some recovery of the device degradation. The fraction of the recoverable component (r) is modeled as shown in equation (2):

$$r = \left(1 + \beta \left(\frac{t_{relax}}{t_{stress}} \right)^{\phi} \right)^{-1} \qquad (2)$$

where β is a scaling parameter, t_{relax} is the total recovery time, t_{stress} is the total stress time and ϕ is a dispersion parameter [24]. Since *r* is the fraction of the total possible recovery remaining, equation (2) can now be used to divide the total degradation during stress from (2) into a recoverable component and a permanent component [19] with the recoverable component given by:

$$\Delta V_{t\_rec} = r \Delta V_{tp} \qquad (3)$$

The recovery was also modeled according to the RD framework. When the stress is released, a recovery phase is initiated and the amount of threshold voltage shift decreases according to:

$$\Delta V_{tr} = \Delta V_{t\_rec} \frac{\zeta_1 + \zeta_2 \exp(-E_a / kT) t_r}{\zeta_3 t} \qquad (4)$$

where $\zeta_{1,2,3}$ are constants which depend on the oxide thickness and the back diffusion rate of hydrogen and temperature, *t* is the total time and t_r is the recovery time and V_{t_rec} is the maximum possible recovery [7],[20] calculated from equation (3).

Unlike NBTI, there is still a lack of a truly scalable physical model for PBTI. However equations (1) and (4) are adequate for capturing the temperature and voltage dependency of the stress and recovery phase of degradation due to PBTI [23],[25] with fitting parameters extracted from [19].

In summary, the device degradation during periods of stress is estimated using equitation (1) while the recovery is estimated using equation (4) enabling accurate cycle-by-cycle tracking of device degradation during system operation.

The degradation of threshold voltage results in longer delays at the circuit level, which eventually result in failure of circuit performances. For any circuit component, a threshold can be determined such that shifts in the threshold voltage results in circuit-level failure, as was demonstrated in [26].

2) *Gate Oxide Breakdown*

Gate oxide breakdown is modeled as a leakage (low resistance) path through the oxide. Several frameworks describing the physics behind the phenomenon of dielectric degradation and breakdown have been proposed. The thermo-chemical model, however, provides a deterministic description of the process of oxide trap generation. It has been shown that the application of electric field on molecular dipole moments lowers the activation energy for bond breakage [8]. This in turn leads to enhanced trap formation in the dielectric. It has been shown [8] that trap generation depends on the electric field is described by an Eyring equation:

$$N_t = \kappa \exp\left(\left(E_a + \kappa_2 E_0\right) \Big/ kT \right) t^{n}, \qquad (5)$$

where κ is the total density of traps available for generation, E_a is the activation energy, k is the Boltzmann's constant, κ_2 is a device constant dependent on the field acceleration factor, and E_0 is the oxide electric field. Traps form throughout the oxide, and when the number of traps exceeds a threshold, a leakage path forms through the oxide.

The formation of a leakage path does not necessarily result in circuit-level failure. However, a threshold can be found when the number of traps links directly to circuit-level failure, as was demonstrated in [2].

IV. Wearout-Aware Compilation

With the high-level system wearout monitoring framework described in the previous section, we can measure and compare the effects of different compilation techniques on the wearout of a register file. Before we describe the actual algorithms that were employed for wearout mitigation, we need to characterize and understand the functioning of a register file and its wearout over time.

With the models from Section 3, the first step in determining the wearout profile of the register file is its division into sectors with similar wearout behavior. Sectors are similar if they undergo the same operations, i.e. a read operation, a write operation, or neither with the current data being retained. Hence, when a register is being read, the transistors undergoing positive/negative voltage bias stress are identified. A similar process is repeated for the write operation. When neither a read or a write occurs, the latches are identified as being under constant DC stress, resulting in worst-case BTI and GOBD damage.

Typically, any wearout mechanism '*X*' can be defined as

$$X = f\left(s, t, T\right) \qquad (6)$$

where '*s*' is the stress profile in terms of the number of reads and/or writes, '*t*' is the timing information associated with the profile '*s*' and the set *T* is the thermal profile. As an example, consider a register that undergoes two writes, and one read at times t_{wr1}, t_{wr2} and t_{rd} during a total run-time of t_r while the temperature variation in the register during t_r is represented by T_r. Then for equation (6), $s=\{writes=2,reads=1\}$, $t=\{t_{wr1}, t_{wr2}, t_{rd}\}$ and $T= T_r$.

The aim of the wearout mitigating algorithms proposed is a compiler-directed spreading of device wearout at runtime. However, at compile-time, we lack the necessary information to derive a complete wearout profile for the wearout mechanisms mentioned in Section 3. Note that in a typical compilation framework, the complete stress profile of the register file is available only after the completion of the compilation process, while the thermal and timing information is available at runtime. As a result it becomes impossible to accurately predict the device wearout at the compilation phase. To overcome this issue, we propose the compilation framework with two phases shown in Figure 2, which are run sequentially.

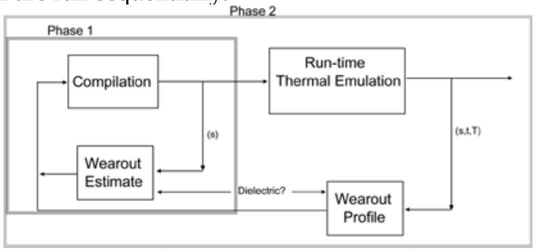


Figure 2: An overview of the proposed compilation framework

Phase 1 consists of a reliability optimization scheme consisting of a compilation module and a wearout estimating module, while phase 2 also takes into account the run-time information and the thermal profile for a more complete wearout profile. The reason for dividing the compilation flow into two phases is the excessive time associated with the derivation of a detailed wearout profile for all wearout mechanisms under study. Hence, to reduce the time associated with the complete compilation flow, a crude first level optimization is done in phase 1. It should be noted here that it took only a few iterations of phase 1 to complete the wearout estimate for our examples while typically only a single run of phase 2 is needed to update the required information from phase 1. The reliability-aware compilation might run for a few minutes, at most, but this is still an insignificant overhead, since the time in phase 1 is negligible compared to the time in phase 2.

The wearout estimating module of phase 1 estimates the degradation due to the wearout mechanisms under study using the stress profile from the compilation module. A constant time window is assigned to each variable and a constant temperature is assumed for each register. The estimate of the total wearout under the current compilation run is then fed into the compilation module which then reassigns the registers depending in the wearout estimate from the last compilation run. Even though a constant temperature is assumed for the wearout models at this point in time, hot spots are taken into account during compilation and avoided by keeping apart registers with high activity levels. In phase 2, the stress timing information and thermal profile of the register file is included in the framework and the detailed wearout profile is used to verify if the required wearout distribution has been achieved or if further optimization is required.

To accommodate register window based architectures, two levels of wearout mitigation, within windows and within registers inside each window, are supported in the proposed compilation framework, as shown in Figure 3.

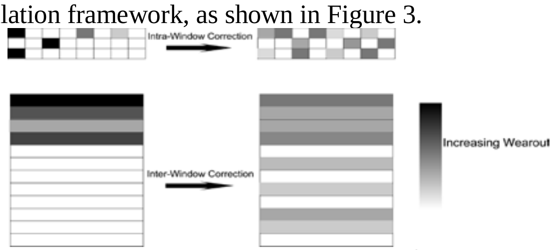


Figure 3: Adaptive stress distribution in the register file. Color intensity represents the amount of wearout.

A. Adaptive Variable Assignment (AVA)

Although this work targets a register-window-based architecture, our first approach for wearout reduction can be just as effective for any other architecture, since the aim here is the adaptive assignment of the active registers, irrespective of whether they are in a window or the whole register file is accessed at once.

Wearout awareness was introduced in the compiler in such a way that the original optimizations employed in the compiler itself are not compromised in any way, while still leaving room for reliability optimization. Typically, for every new variable assignment to registers, the compiler accesses the color graph and finds a set of registers that are available for assignment [6]. The flow is then transferred to a series of filters, as shown in Figure 4. These filters introduce further reductions in the set of available registers, according to some pre-determined optimization criteria, resulting in what is known as the reduced availability set (Sr). Once the reduced availability set is formed, typically a compiler picks the first register from the list, making no distinction between the registers in this set. Hence, in the case of a reduced availability set larger than one, there is still room for the introduction of improved register assignments for reliability.

The proposed reliability filtering process can be introduced before or after the reduction of the set to Sr depending on how reliability critical the target application is. Introducing the reliability filter before reduction of the availability set gives the highest precedence to reliability optimization.

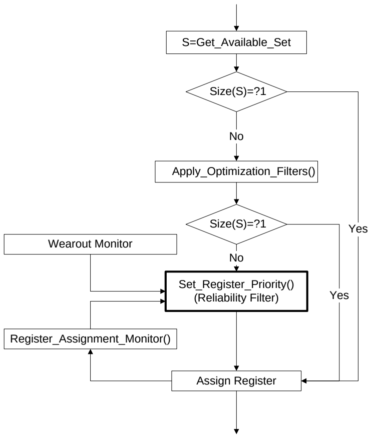


Figure 4: The register assignment flow with the additional reliability filter included.

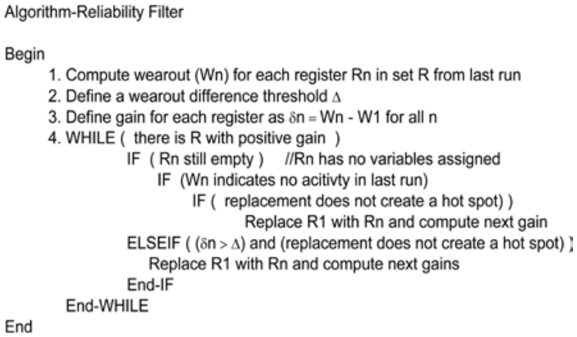
As can be seen, without extensive changes in the compiler flow, we have inserted a new reliability filter in the register assignment flow. Figure 5 presents the algorithm implemented in the reliability filter shown in Figure 4. The reliability filter assigns a gain value to all the registers in the reduced set according to the input from the register wearout monitor, which keeps track of the register wearout from the last run. The gain δn of a register at position Rn is the difference between its computed wearout Wn and the wearout of the register at position 1. Wn was computed using a weighted sum of the parameters associated with the wearout mechanisms under study, i.e.

$$\delta n = Wn - W1 \tag{7}$$

$$Wn = \Gamma_{NBTI} V_{tPMOS}(s,t,T) + \Gamma_{PBTI} V_{tNmos}(s,t,T) + \Gamma_{GOBD} N_t(s,t,T)$$

where Γ_X is the weight assigned to wearout due to mechanism X. Hence, a higher weight for NBTI means a higher precedence is given for NBTI reduction during compilation.

Variable assignments for each register are also tracked in the register assignment monitor block shown in Figure 4. Details are shown in Figure 5. The algorithm starts with a set of registers R for variable assignment. R is the reduced availability set. In assigning registers to variables, precedence is given to registers that are yet to be assigned any variables and whose wearout results indicate a lack of activity from the last run. This forces the spreading of activity across the register file. For the case when no such register is empty, i.e. has previously not been assigned variables, the wearout of the first register in the set of available registers is compared with the wearout of all the other registers and the gain for each register is computed. The first register in the set is then replaced if the gain of the current register is greater than a predetermined value Δ. However, a replacement is only executed if it does not result in a *hot spot.* A register is labeled as a hot spot if it or any of its surrounding registers exceeds a pre-determined number of variable assignments.



```
Algorithm-Reliability Filter

Begin
        1. Compute wearout (Wn) for each register Rn in set R from last run
        2. Define a wearout difference threshold $\Delta$
        3. Define gain for each register as $\delta n = Wn - W1$ for all n
        4. WHILE (  there is R with positive gain  )
                    IF  ( Rn still empty )     //Rn has no variables assigned
                        IF  (Wn indicates no acitivty in last run)
                            IF (  replacement does not create a hot spot) )
                                  Replace R1 with Rn and compute next gain
                    ELSEIF ( ($\delta n > \Delta$) and (replacement does not create a hot spot) ]
                       Replace R1 with Rn and compute next gains
                    End-IF
           End-WHILE
End
```

Figure 5: Algorithm for reliability filter with time complexity of O(n).

In summary, the algorithm passes through the register set of length *n* once, and replaces the first register if it satisfies the conditions mentioned above. At the end of the pass, we have the best candidate for register assignment in position 1 in the set. The algorithm has a time complexity of O(n). Once the variable is assigned to a register, the register assignment monitor and the hot spot labels are updated and the process is repeated for the next variable.

B. *Variable Multi-Window Context Switching*

The new AVA is limited by the fact that it can only monitor and select registers, for variable assignment, from the set of registers visible to the compiler at that instant. This would be of no consequence if the compiler had access to the whole register file for every register/variable assignment. This, however, is not the case for register window based architectures, such as SPARC, where the register file is divided into smaller sets. Hence, when assigning a variable, the compiler can only access a single window.

Therefore, we propose a variable multi-window context switching (VMWCS) scheme which balances the switching activity among the register windows in the register file.

In general, a program includes a number of functions (or procedures) that are called in a hierarchal manner, where the *main* function is at the top level of the hierarchy and uses register-window *i=1* in execution. All the functions found at the same level of the hierarchy use the same register-window, which is sequentially next to the one used by the ancestor functions.

Such methods of assigning register windows are inefficient in terms of reliability and thermal behavior, since they lead to highly uneven activity levels among the windows, i.e. the levels in the hierarchy are not equally weighted in terms of switching activity. This leads to an uneven distribution of electrical stress throughout the register file and may result in high temperatures due to the close proximity of highly-accessed register windows. Thus VMWCS balances the inter-window stress profile by studying the structure of the program, by exploring the functional interdependencies, and by assigning various register windows to functions at the same hierarchal level at compiler preprocessing time.

Figure 6 shows the flow chart of VMWCS. The proposed scheme executes in three main stages, (1) information extraction, (2) windows setting, and (3) window assignment.

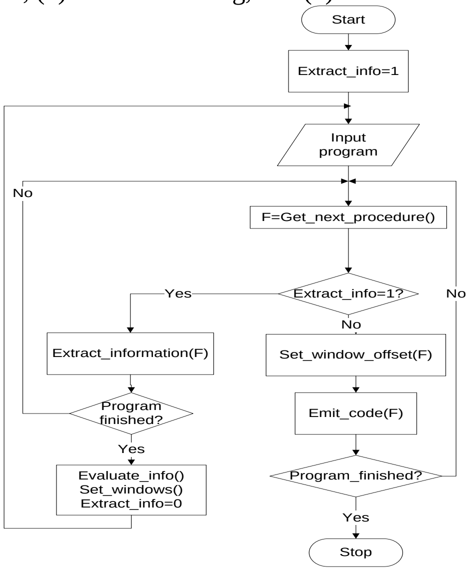


Figure 6: The proposed variable multi-window context switching scheme flow.

The information extraction phase is performed initially by setting a flag, named *Extract_info,* to 1. At this stage, all the necessary information needed to construct a hierarchal tree of the program (functional calls and access weight for each procedure) is extracted and exported to files to be used later (left branch in Figure 6). Then, in the windows setting stage, the exported data is used for the generation of the hierarchal tree and the subsequent initialization of a factor, called weight, for each function (*F_P)* which is calculated using:

$$weight(F_P) = Access(F_P) \times \prod_{i=2}^{P} repeat(F_i) \qquad (8)$$

where *Access* is the extracted access weight of the current function, which is the calculated degradation factor based on the function utilization of a register window. The factor *repeat* is an estimate of the number of calls to the function F_i. We get an estimate of the number of function calls from the number of iterations in which this function call is embedded and the number of explicit calls to this function. It is important to note that we extract these values at compile time. Although runtime profiling of these functions is an adequate methodology [27], our approach is more time efficient, and the estimated number of calls at compile time is close to the actual number of calls at runtime. Once calculated, this weight factor is then used to select the window for each function, with the aim of uniformly distributing the wearout across the register file.

Based on the impact of each function on register window wearout, a number *k*, the window step size, is assigned to each function. *k* represents the step size relative to the window of the immediate ancestor of the function F_P (right branch in Figure 6). Thus, the output from this stage is a list of all the functions and their assigned window step sizes.

The list generated by the window setting stage is then passed onto the window assignment stage, which generates the instructions required to shift a function from one window to another. Therefore, depending on the window step sizes, the window assignment stage then lays out the functions across the register file in such a way as to minimize wearout during runtime (right branch in Figure 6).

VMWCS has a computational complexity which is related to two main factors, the number of available windows (N_w) and the number of functions or procedures in the compiled program (N_f). The algorithm starts with computing the average degradation of the register file (Deg_W) if the function degradation impacts are equally distributed between the windows. From this value, the algorithm passes through each window and allocates a number of functions from the available set, such that the overall degradation impact is close to Deg_W. The computational complexity of VMWCS is $O(N_w.N_f)$.

VMWCS introduces a few additional instructions and some compile time overhead. The instruction overhead is translated into an increase in code size and execution time. However, the overhead has a negligible impact, as we show in Section 5.

V. Case Study: SPARC V8

The SPARC V8 is a RISC based 32-bit architecture with as many as 128 general purpose registers. This window based architecture was selected as a case study for this work. The windowing nature of this architecture allows us to evaluate the effectiveness of both the AVA and the VMWCS wearout mitigation schemes.

At any point, only 32 registers are visible to the software. Out of these, eight are global, while the rest of the 24 form the register window. This register window moves up and down the register stack with every function call or return. Each window has eight local registers, with 16 being shared with the neighboring windows, with the aim of passing data between function calls.

The viability of the proposed wearout mitigation scheme was assessed across a variety of benchmarks using the HW/SW emulation platform presented in [18]. While the AVA was used for wearout balancing inside the active window, VMWCS preformed the inter-window wearout balancing. NBTI, PBTI and gate-oxide degradation were monitored for each benchmark, and the results for the proposed reliability aware compilation scheme, presented in this work, were compared to the default compilation scheme [6] and the thermal aware compilation scheme presented in [4]. The comparison with [4] enables us to determine the impact of stress on lifetime, separately from temperature.

BTI degradation was monitored using the Vt drift, while gate-oxide degradation was monitored using the approximate oxide traps generated.

Let's first consider a detailed analysis of the effect of the proposed scheme on NBTI and GOBD for one of the benchmarks, FFT, which uses a limited number of windows. Analysis for PBTI was left out since PBTI is similar in behavior to NBTI. Figure 7 presents the normalized difference between the threshold voltage degradation of PMOS devices in the register file for the default compilation scheme [6] and the proposed reliability aware compilation scheme, i.e.,

$$Z = \frac{(Vt\_default) - (Vt\_reliability)}{(Vt\_default)} \qquad (9)$$

where Z represents the average wearout of the devices in a single register after the completion of a single run of the benchmark. A positive value indicates a improvement as compared to the default scheme. Figure 7 indicates that some registers show clear improvement in mitigating NBTI by as much as 10% during runtime. However, this comes at the cost of an increase in the degradation of other registers. The difference between the magnitude of improvement at some locations and the magnitude of the increase in wearout in others is due to the improvement in the thermal profile of the register file at run-time and the recoverable nature of NBTI.

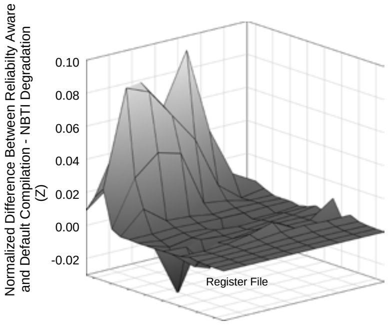


Figure 7: The difference between Vt degradation for PMOS devices between the reliability aware compilation and the default compilation method for FFT. The x-y plain represents the physical layout of the register file.

Compared to BTI, gate-oxide degradation is not recoverable, and hence there is a direct tradeoff between reducing the wearout in one region of the register file vs. another, as seen in Figure 8. However, overall, the register file showed improvement in gate oxide degradation during runtime.

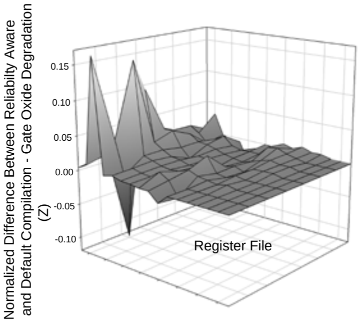


Figure 8: The difference between gate-oxide degradation for PMOS devices between reliability aware compilation and the default compilation method for FFT. The x-y plain represents the physical layout of the register file.

The compilation process keeps track of wearout of registers, on top of thermal-awareness, making our scheme a truly reliability aware scheme. Figures 9-11 plot the amount of degradation due the proposed reliability-aware and thermal-aware schemes normalized against the default compilation scheme. Hence, for example, a value of 0.8 on these figures indicates a 20% improvement for a specific reliability mechanism over the default compilation scheme. Our results indicate that we were able to achieve significant improvement over the thermal-aware compilation method across different benchmarks.

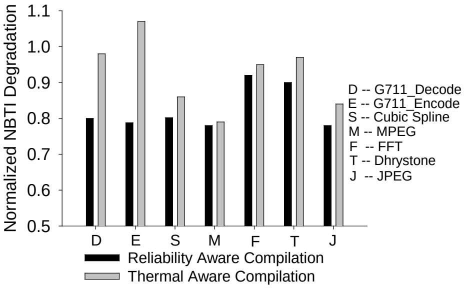


Figure 9: NBTI degradation across different benchmarks for reliability-aware and thermal-aware compilation normalized against the default compilation mode.

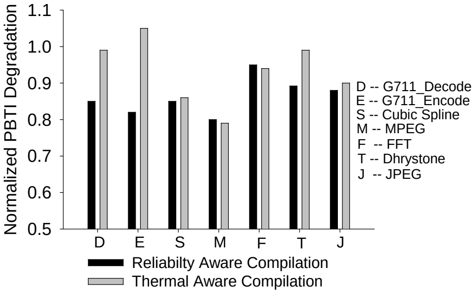


Figure 10: PBTI degradation across different benchmarks for reliability-aware and thermal-aware compilation normalized against the default compilation mode.

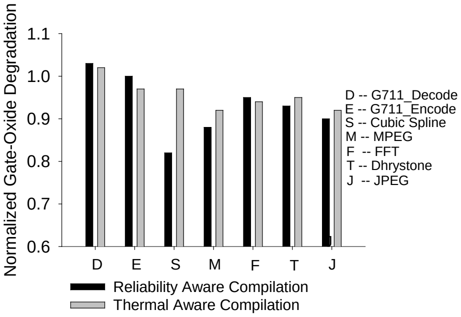


Figure 11: Gate-Oxide degradation across different benchmarks for reliability-aware and thermal-aware compilation normalized against the default compilation mode.

Figure 9 indicates the increase in PMOS threshold voltages due to NBTI normalized against the increase in PMOS threshold voltages due to the default compilation scheme i.e. $\frac{Vt\_relaibility/thermal}{Vt\_default}$. On average, NBTI was significantly reduced for the reliability-aware scheme with around 15-20% improvement across the benchmarks. Compared to that, thermal-aware compilation [4] did not show any clear trend, with the MPEG showing significant improvement, while Encode experiencing worsened NBTI. The significant improvement in the NBTI degradation for the reliability aware scheme is due to the healing nature of NBTI.

Stress balancing across the register file helps in distributing the workload throughout the register file, hence reducing the number of completely inactive registers, which result in worst case NBTI degradation for the latches. The fact that more devices in the register file experience periods of no-stress helps in the recovery of some of the degraded threshold voltages for these devices. The thermal-aware scheme failed to show any significant improvement for benchmarks where register assignment had little impact on the thermal profile [4] (Decode and Encode), while the comparatively better results for the reliability-aware scheme for the two can be solely attributed to stress balancing.

Similarly, Figure 10 shows the results for NMOS degradation due to PBTI across different benchmarks. Due to their similar nature, PBTI followed a similar trend to PMOS degradation due to NBTI. Our results indicate slightly better results for NBTI as compared to PBTI. However, it should be noted that the relative magnitudes of NBTI and PBTI degradation and their recovery is a strong function of the choice of dielectric.

Gate oxide degradation, with its non-healing nature, also showed encouraging results across the benchmarks. However the improvement in gate oxide degradation was the least among the three mechanisms considered. Gate oxide degradation is also the hardest to predict, since not only does the amount of degradation depend on the activity intensity, but also on the type of activity, with a cell being thrice toggled having a possibility of both, higher degradation and lower degradation, in comparison with a cell that is toggled twice, depending on the arrangements of the toggles with respect to time.

Table 1 summarizes our results across different benchmarks for the presented reliability-aware scheme presented in this work and the thermal-aware scheme of [4].

Additionally it also compares the increase in code size and compile-time for both schemes. Hence, this table shows that the proposed reliability-aware scheme significantly improves register-file lifetime by combining stress and thermal balancing, while having a limited code overhead increase. Although the increase in compile-time is significant, it is in fact negligible in comparison with the program lifetime, where program lifetime is calculated as a single compile-time and numerous execution-times.

Table 1: Improvement in different wearout mechanisms and increase in code size for reliability-aware and thermal-aware compilation schemes.

Compilation Scheme	Code Size %	Compile time	NBTI	PBTI	GOBD
Proposed reliability-aware method	0.19%	120%	20%	14%	7%
Thermal-aware method [4]	0.113%	20%	7%	9%	4%

VI. Conclusions

A compilation scheme with reliability as the optimization criteria has been presented. We have shown how thermal-awareness alone might be inadequate for enhancing lifetime. Our proposed scheme makes use of both thermal balancing and stress balancing to fully optimize compiler-directed register file assignment for lifetime improvement. Experimental results show that our proposed compilation scheme achieves better reliability improvement with respect to the state-of-the-art thermally-aware compilation techniques for BTI and gate-oxide degradation.

Acknowledgements

The authors would like to acknowledge the support for this work of ACE Associated Compiler Experts by providing the complete license to use the CoSy compiler development system at ESL-EPFL, as well as the helpful insights on the modification of the CoSy compiler chain.

References

[1] S. Borkar, "Designing reliable systems from unreliable components: the challenges of transistor variability and degradation," *IEEE Micro*, vol. 25, pp. 10-16, 2005.

[2] F. Ahmed and L. Milor, "Reliable cache design with on-chip monitoring of NBTI degradation in SRAM cells using BIST," in *28th VLSI Test Symposium (VTS)*, 2010, pp. 63-68.

[3] T. Kim, et al., "Silicon odometer: an on-chip reliability monitor for measuring frequency degradation of digital circuits," in *IEEE Symposium on VLSI Circuits*, 2007, pp. 122-123.

[4] M. Sabry, et al., "Thermal-aware compilation for system-on-chip processing architectures," in *Proceedings of the 20th symposium on Great lakes symposium on VLSI*, 2010, pp. 221-226.

[5] J. Srinivasan and S. Adve, "Predictive dynamic thermal management for multimedia applications," in *Proceedings of the 17th annual international conference on Supercomputing*, 2003, pp. 109-120.

[6] "ACE Cosy Compiler. http://www.ace.nl/compiler/cosy.html.."

[7] M. Alam and S. Mahapatra, "A comprehensive model of PMOS NBTI degradation," *Microelectronics Reliability*, vol. 45, pp. 71-81, 2005.

[8] D. Qian and D. Dumin, "The Electric Field, Oxide Thickness, Time and Fluence Dependences of Trap Generation in Silicon Oxides and Their Support of the E-model of Oxide Breakdown," in *Proceedings of the 1999 7th International Symposium on the Physical and Failure Analysisof Integrated Circuits*, 1999, pp. 145–150.

[9] W. Wang, et al., "An efficient method to identify critical gates under circuit aging," in *Proceedings of the 2007 IEEE/ACM international conference on Computer-aided design*, 2007, pp. 735-740.

[10] J. Srinivasan, et al., "Lifetime reliability: Toward an architectural solution," *IEEE Micro*, vol. 25, pp. 70-80, 2005.

[11] Y. Lee, et al., "Managing Bias-Temperature Instability for Product Reliability," in *International Symposium on VLSI Technology, Systems and Applications, VLSI-TSA* 2007, pp. 1-2.

[12] A. Calimera, et al., "NBTI-Aware power gating for concurrent leakage and aging optimization," in *Proceedings of the 14th ACM/IEEE international symposium on Low power electronics and design*, 2009, pp. 127-132.

[13] D. R. Bild, et al., "Minimization of NBTI performance degradation using internal node control," in *Proceedings of the Conference on Design, Automation and Test in Europe*, 2009, pp. 148-153.

[14] H. Yun and J. Kim, "Power-aware modulo scheduling for high-performance VLIW processors," in *Proceedings of the 2001 International Symposium on Low Power Electronics and Design*, 2001, pp. 40-45.

[15] D. Sylvester, et al., "Elastic: An adaptive self-healing architecture for unpredictable silicon," *IEEE Design & Test of Computers*, vol. 23, pp. 484-490, 2006.

[16] B. Hamidzadeh, et al., "Dynamic scheduling techniques for heterogeneous computing systems," *Concurrency: Practice and Experience*, vol. 7, pp. 633-652, 1995.

[17] N. Aggarwal, et al., "Configurable isolation: building high availability systems with commodity multi-core processors," in *Proceedings of the 34th annual international symposium on Computer architecture*, 2007, pp. 470 - 481.

[18] D. Atienza, et al., "HW-SW emulation framework for temperature-aware design in MPSoCs," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 12, pp. 1-26, 2007.

[19] T. Grasser, et al., "Simultaneous Extraction of Recoverable and Permanent Components Contributing to Bias-Temperature Instability," in *IEEE International Electron Devices Meeting.* , 2007, pp. 801-804.

[20] S. Bhardwaj, et al., "Predictive modeling of the NBTI effect for reliable design," in *Custom Integrated Circuits Conference, CICC*, 2006, pp. 189-192.

[21] L. Xiaojun, et al., "Compact Modeling of MOSFET Wearout Mechanisms for Circuit-Reliability Simulation," *IEEE Transactions on Device and Materials Reliability*, vol. 8, pp. 98-121, 2008.

[22] S. Pae, et al., "BTI reliability of 45 nm high-K + metal-gate process technology," in *IEEE International Reliability Physics Symposium.*, 2008, pp. 352-357.

[23] L. Sun-Me, et al., "Effects of BTI during AHTOL on SRAM V<inf>MIN</inf>," in *IEEE International Reliability Physics Symposium (IRPS)* 2011, pp. 1-6.

[24] S. Ramey, et al., "Frequency and recovery effects in high-&#x03BA; BTI degradation," in *IEEE International Reliability Physics Symposium.*, 2009, pp. 1023-1027.

[25] S. K. Krishnappa and H. Mahmoodi, "Comparative BTI reliability analysis of SRAM cell designs in nano-scale CMOS technology," in *12th International Symposium on Quality Electronic Design (ISQED)*, 2011, pp. 1-6.

[26] F. Ahmed and L. Milor, "NBTI resistant SRAM design," in *4th IEEE International Workshop on Advances in Sensors and Interfaces (IWASI)*, 2011, pp. 82-87.

[27] X. Zhou, et al., "Temperature-aware register reallocation for register file power-density minimization," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 14, pp. 1-26, 2009.