Processor Reliability Enhancement through Compiler-Directed Register File Peak Temperature Reduction

Chengmo Yang and Alex Orailoglu
Computer Science and Engineering Department
University of California, San Diego
9500 Gilman Drive, La Jolla, CA 92093
{c5yang, alex}@cs.ucsd.edu

***Abstract*—Each semiconductor technology generation brings us closer to the imminent processor architecture heat wall, with all its associated adverse effects on system performance and reliability. Temperature hotspots not only accelerate the physical failure mechanisms such as electromigration and dielectric breakdown, but furthermore make the system more vulnerable to timing-related intermittent failures. Traditional thermal management techniques suffer from considerable performance overhead as the entire processor needs to be stalled or slowed down to preclude heat accumulation. Given the significant temporal and spatial variations of the chip-wide temperature, we propose in this paper a technique that directly targets one of the resources that is most likely to overheat in current processors, namely, the register files. Instead of duplicating or physically distributing the register file, we suggest to attain power density control through exploiting the extant spatial slack associated with register file accesses. Based on application-specific access profiles, a compiler-directed register shuffling strategy is proposed to deterministically construct the logical to physical register mapping in a rotating manner. Simulation results confirm that the proposed technique attains, within a limited hardware budget and negligible performance degradation, effective reduction in peak temperature and hence in the expected fault rates for the entire chip.**

I. Introduction

Advances in IC fabrication processes have enabled dramatic numbers of transistors to be integrated on a single chip, in the process though unfortunately bringing us ever closer to the imminent processor architecture heat wall. The continuous scaling of circuit current, clock speed and device density further lead to significant temperature increases that adversely degrade the performance and reliability of the chip.

Previous work shows that a higher temperature accelerates the chemical processes taking place inside the chip, thus making the system much more vulnerable to various failure mechanisms such as electromigration, stress migration and dielectric breakdown [1]. A mere 10 – 15$°C$ rise in the operating temperature could halve the life span of the circuit [2]. Meanwhile, temperature hotspots also elevate the amount of intermittent and transient faults that may occur during execution. A higher temperature reduces the mobility of the charge carriers, thus diminishing the switching speed of the transistors. Negative bias temperature instability (NBTI) and hot carrier injection (HCI) furthermore cause violations of circuit timing constraints [3]. As reported in [4], delay fault rates double for each 10$°C$ increase in temperature. Moreover, as every 20$°C$ increase in temperature causes a 5-6% increase in Elmore delay in interconnects, clock skew problems become noticeable for temperature spatial variations of around 20$°C$ and above [5]. These performance and reliability issues are further worsened by the positive feedback loop between temperature and leakage power; not only are the two positively correlated but leakage current is furthermore exponentially related to temperature, exacerbating further the effects of the positive feedback.

Traditional packaging and cooling solutions typically need to target worst case peak temperature, resulting in extremely expensive packaging solutions as prevailing temperature levels rise (approximately $10 per Watt above 65$°C$). To keep the chip-wide temperature within the thermal capacity of the cooling package, system-level *Dynamic Thermal Management* (DTM) techniques become indispensable for both high-end general purpose processors [6] and low-power embedded cores [7], [8]. These DTM techniques, such as clock gating [9], fetch toggling [10], and dynamic frequency and voltage scaling [6], control worst-case temperature through globally stalling or slowing down the computation of an overheated core, thus imposing significant performance deterioration.

Due to its high utilization (accessed 2–3 times per instruction) and relatively small area, the *register file* has been established as one of the hardware units most likely to overheat in current processors [6]. This localized "hotspot" can reach critical temperature levels regardless of average or peak external package temperature, thus ending up constraining the overall performance and reliability of the whole chip. More crucially, due to the fact that 90% of the execution time is spent on loops where only a small subset of registers is repetitively accessed, register file accesses also exhibit high asymmetry during program execution. This asymmetric register utilization furthermore leads to considerable temperature differentials, since most of the heat generated within a microarchitectural block is dissipated vertically to the heat sink rather than laterally to adjacent blocks [6].

The aforementioned register file access characteristics indicate that the peak temperature within a register file, the hottest spot of a modern processor, can be effectively controlled through minimizing the peak power density across the register file, which in turn can be accomplished through distributing the accesses uniformly throughout the register file. To achieve this goal, the register names obtained at the decode stage cannot be directly used to access the register file, as they would result in a highly asymmetric access distribution. Instead, we propose in this paper an *iteration-based* register shuffling technique which, through *physical remapping of heavily accessed logical registers prior to local heat buildup*, effectively controls the

register file peak temperature. Moreover, to minimize the hardware overhead and hence the extra power dissipation of this dynamic remapping support, this register name shuffling process is *deterministically* directed by the compiler. A post-compilation adjustment of the register names allows *regularity* to be embedded within register accesses, so that accesses to each register can be evenly balanced across loop iterations with no need of any hardware mapping table to keep track of register usage or register mapping information. This extremely low hardware overhead therefore enables an easy incorporation of the proposed technique into low-power embedded processors to attain temperature control.

The rest of the paper is organized as follows. Section II briefly reviews the related work. Section III outlines the technical motivation. Sections IV and V present in detail the proposed register shuffling scheme and the corresponding implementation, respectively. The efficacy of the proposed technique is experimentally verified in section VI. Finally, section VII offers a brief set of conclusions.

II. Related work

The adverse impact of operating temperature on system reliability has been studied extensively. Researchers have built both analytical and experimental models for temperature-induced fault rate increases, such as delay violations [4], negative bias temperature instability [3], neutron-induced latchup [11], and on-chip interconnect [5]. A number of design and modelling challenges have been summarized in [12].

Based on the physical and power consumption characteristics of each hardware resource, researchers have developed architectural thermal models, such as the *HotSpot* [6], for calculating transient temperature response. Using these models, two types of architectural *Dynamic Thermal Management* (DTM) techniques have been proposed to prevent the chip from reaching critical temperature levels. *Temporal* techniques slow down heat accumulation either at fine granularity through fetch toggling [10], decode throttling [13], frequency and voltage scaling [6], or at coarse granularity through periodically stopping the computation to induce cooling [14]. Obviously, slowing down or stopping the entire computation engenders significant performance degradation. *Spatial* techniques, on the other hand, reduce temperature by shifting computation from a hot resource to a relatively cool resource, at the granularity of functional units [15], pipelines [15], execution clusters [16], or even cores on a single chip [17]. These spatial techniques sizably reduce the performance overhead associated with temporal DTM techniques, yet their applicability requires the existence of spare or underutilized resource copies.

Since the register file is one of the most power-hungry and overheated resources in current processors, increasing research attention has also been paid to the design of thermal-aware register files. Gate level techniques, such as single- or multi-level banking [18] and bit-partitioning [19], have been proposed to reduce the power consumed in each register file access and, hence, the peak temperature. As temperature is determined not only by power density but furthermore by heat dissipation speed, researchers have also proposed to either incorporate an extra register file [6] to increase the average idle time, or physically distribute the register file into multiple clusters [16] to accelerate heat dissipation. Duplication or distribution of critical resources such as the register file though engenders sizable increases in chip area, design and wiring complexity, as well as critical path delay, thus significantly limiting their applicability.

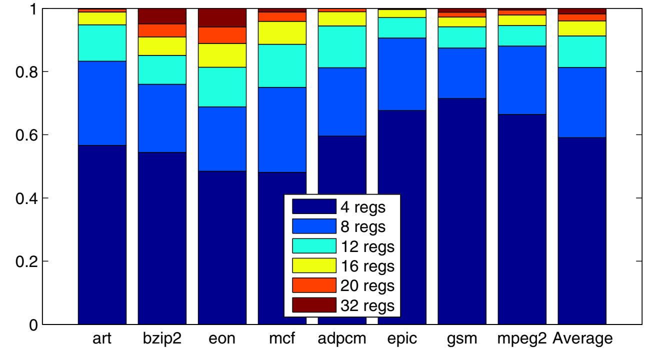

Fig. 1. Cumulative register access ratio

As the peak temperature is determined by the most frequently accessed registers, thermal-aware register reassignment techniques [20], [21] have been proposed. Both techniques reduce the level of asymmetry in register accesses through mapping the most frequently accessed registers to distinct register banks. However, as both techniques need to revisit the NP-hard problem of register allocation, the quality of the solutions is determined by the quality of their heuristic algorithms. More crucially, no matter how good the heuristics are, such techniques cannot completely eliminate the access asymmetry to each register, since such asymmetry directly derives from the asymmetric variable utilization of the program. As a result, these techniques can only attain a coarse-grained *spatial* balance, implying that they are only applicable to multi-bank register files. In contrast, the technique we propose iteratively maps a logical destination to distinct physical registers across loop iterations, thus enabling the attainment of a fine-grained *temporal* balance to each individual register without revisiting the register allocation problem.

III. Technical Motivation

The design of a dynamic register shuffling process to reduce the register file peak temperature and, hence, to improve chip reliability, is motivated by the observation that the code generated by the compiler exhibits highly asymmetric register access activity. A traditional register allocation scheme in a temperature-unaware compiler initially assumes an infinite set of *virtual registers* for representation, and subsequently maps these virtual registers into a fixed number of *architectural registers*. The decision regarding which physical registers in particular are to be allocated, however, does not take into consideration the access distribution, thus leading to a highly asymmetric register access activity. Figure 1, which presents the cumulative register access ratios of a set of *SPEC2000* (shown in the first 4 bars of Figure 1) and *mediabench* programs (the second 4 bars), provides experimental confirmation. As can be seen, both sets of benchmarks exhibit an appreciable amount of imbalance in that 48% to 71% of the total register accesses are

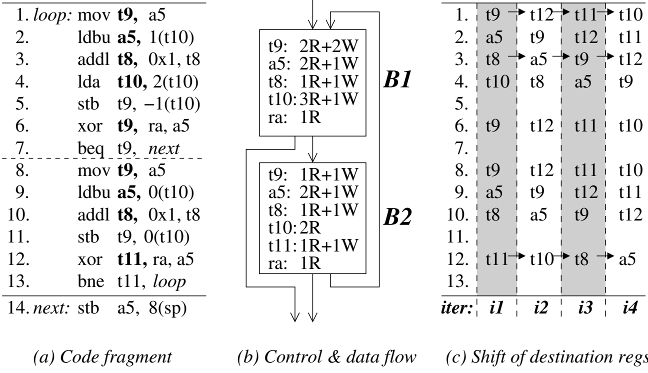

Fig. 2. A loop example obtained from *bzip2*

to 4 registers. *Mediabench* programs display a higher amount of imbalance as compared to *SPEC2000* benchmarks. On the average, a set of 12 out of a total of 32 registers is able to capture more than 90% of the total register accesses.

The asymmetric register utilization shown in Figure 1 confirms that by evenly distributing the accesses to each individual register within frequently executed loops, the peak power density and hence the peak temperature can be effectively reduced. However, such a balancing cannot be directly attained through an assignment of register names during code generation, as the access asymmetry directly derives from the asymmetric variable utilization of the program. This limitation can be more concretely illustrated by a representative code fragment presented in Figure 2a, an unrolled loop composed of 13 instructions that accounts for more than 25% of the total execution time of *bzip2*. The corresponding control flow and register access pattern are presented in Figure 2b. As can be seen, this loop exhibits a quite unbalanced register utilization as it only accesses 6 (a_5, t_8–t_{11}, ra) out of 32 general purpose registers. Among these 6 registers, a_5, t_9, and t_{10} are accessed most frequently. While the compiler may be able to use more registers by separating multiple definitions of a single register (a_5 and t_9) or further unrolling the loop, individual register accesses would still remain skewed due to the existence of singly assigned yet frequently referred registers, such as t_{10}.

Since the static register allocation process cannot completely balance the accesses to each individual register, a dynamic mapping needs to be established between the encoded register names, denoted as the *logical registers*, and the register instances in the register file, denoted as the *physical registers*, to physically remap heavily accessed logical registers prior to local heat buildup. This task could be achieved through using a hardware *mapping table*, such as the one used in conventional superscalar processors, to record free physical registers and redirect register accesses. Unfortunately, such a mapping table imposes a notable amount of hardware complexity, energy consumption and performance overhead. More crucially, as the table needs to be accessed using logical register names at a frequency no lower than that of register file accesses, this small hardware unit itself would hence become a temperature "hotspot" with skewed access distribution. This significant energy and heat overhead confirms that a temperature-aware register file should be able to evenly distribute accesses to each register with **no** reliance on a hardware mapping table. Moreover, as no hardware mapping table is used to keep track of run-time register usage or register mapping information, *regularity* needs to be embedded within register accesses so that the mapping between logical and physical register names can be controlled in a *deterministic* manner. Specifically, the following two tasks need to be accomplished:

- Select a free physical register for each write access *with no dynamic register usage information*;
- Redirect each read access to the corresponding physical register *with no dynamic mapping information*.

These issues necessitate an easily computable dynamic mapping that deterministically controls the renaming process. To attain this goal, in this paper we propose a *compiler-directed* dynamic register shuffling technique to control the logical-to-physical register mapping on a per iteration basis. By exploiting the fact that no fixed, preordained correspondence exists between program variables and register names, the compiler can establish a certain property between the fixed static name and the iteratively varying dynamic name of a logical register, thus enabling the hardware to redirect register accesses with no reliance on a mapping table.

IV. Deterministic Register Shuffling

A. An illustrative example

The fundamental goal of the proposed register shuffling technique is to evenly balance register accesses across loop iterations with no reliance on a hardware mapping table. Accordingly, not only a *dynamic* logical-to-physical register mapping needs to be established, but furthermore *determinism* needs to be embedded within such a dynamic mapping.

To attain a deterministic register remapping, the proposed technique exploits the fact that during loop execution, the physical register used at the *preceding iteration* becomes free whenever a new physical register is allocated as the destination of an instruction. Taking the code fragment presented in Figure 2a as an example, upon a new iteration, if a free register (t_{12} for example) is used as the destination of instruction 1, the old destination, t_9, becomes free thereafter. As a result, t_9 can be used as the new destination of instruction 2, which in turn frees up a_5, allowing it to be used for instruction 3, and so on and so forth. This shift in register assignments can be clearly seen in the ***i2*** column of Figure 2c. Finally, at the end of this iteration, t_{11} has been freed, thus allowing it to be used as the new destination of instruction 1 at the next loop iteration. The remapping of the destination registers during the first 4 consecutive iterations is summarized in Figure 2c.

It can be seen from the register names presented in Figure 2c that the proposed register remapping process exhibits the following two properties:

- Across loop iterations, a logical register is sequentially mapped to all the physical registers before it shuffles back to the initial mapping.
- Within a single iteration, all the assignments of a single logical register are mapped to the same physical register, thus establishing a one-to-one mapping between logical and physical register names.

The first property indicates that the proposed technique can effectively balance accesses to individual registers across loop iterations. Although this technique does not reduce the energy consumed in each register access, it still effectively prevents local heat buildup since heavily accessed logical registers, such as $a5$, t_9 and t_{10} in the example, are mapped to distinct physical registers across loop iterations. As temperature takes at least 0.1 million cycles to rise by 0.1 $°C$ [6], this balanced access activity, achieved at the granularity of loop iterations, enables an effective reduction of the register file peak temperature.

The second property enables the proposed remapping scheme to attain access determinism. Specifically, within a single iteration as a one-to-one mapping is established between logical and physical register names, the dynamic name of a register to be completely determined according to the static name and the loop iteration count, thus eliminating the necessity of a hardware table to record dynamic register mapping.

TABLE I
THE USE OF THE TWO SHUFFLE FUNCTIONS TO SHIFT REGISTER NAMES IN THE *bzip2* EXAMPLE, $B = 1, O = 1, T = 7$

	Modulo Addition							$GF(2^3)$ Multiplication[a]						
	i_0	i_1	i_2	i_3	i_4	i_5	i_6	i_0	i_1	i_2	i_3	i_4	i_5	i_6
$t_9 \rightarrow B$	1	2	3	4	5	6	7	1	2	4	3	6	7	5
$a_5 \rightarrow B^1_{O'}$	7	1	2	3	4	5	6	5	1	2	4	3	6	7
$t_8 \rightarrow B^2_{O'}$	6	7	1	2	3	4	5	7	5	1	2	4	3	6
$t_{10} \rightarrow B^3_{O'}$	5	6	7	1	2	3	4	6	7	5	1	2	4	3
$t_{11} \rightarrow B^4_{O'}$	4	5	6	7	1	2	3	3	6	7	5	1	2	4
$DN(R^i_k)$	$= (SN(R_k) + i)\%7$							$= SN(R_k) \otimes 2^i$						
$SN(R_k)$	$= (SN(R_0) - k)\%7$							$= SN(R_0) \otimes 2^{7-k}$						

[a]The field generating polynomial of $GF(2^3)$ is $x^3 + x + 1$.

B. Destination register name adjustment

At each iteration, the proposed scheme deterministically remaps the kth logical destination register to the physical register used as the $(k-1)^{th}$ destination in the last iteration. This recursive relationship can be formalized as follows, with $DN(R^i_k)$ denoting the dynamic name of the kth destination register at iteration i:

$$DN(R^i_k) = DN(R^{i-1}_{k-1}) = DN(R^{i-2}_{k-2}) = ... \quad (1)$$

Equation (1) illustrates a crucial property of the proposed register shuffling technique: during loop execution all the logical destination registers are iteratively mapped to the same set of physical registers in the same *shifting order*. The shifting order of the *bzip2* example, represented by the arrows in Figure 2c, is (t_9, t_{12}, t_{11}, t_{10}, t_8, a_5, t_9). Moreover, this shifting order happens to be the **reverse** of the order in which each logical register appears as a destination within the loop body.

The *reverse-order* property indicates that if a fixed offset has been imposed between any two *consecutive yet distinct destination register names*, any two *consecutive mappings of a logical register* would also exhibit a fixed offset. More formally, by imposing a fixed offset of O' between the static names of the kth and the $(k-1)^{th}$ destination registers, the dynamic name of register R_k at iteration i, denoted as $DN(R^i_k)$, can be generated through shuffling $DN(R^{i-1}_k)$ by a fixed offset of O. Using V^α_O to denote the shuffle of a value V by an offset O for α times, this fixed-offset relationship can be formalized into the following equations, with O and O' being complements in that $(V^\alpha_O)^\alpha_{O'} = V$ for any positive integers V and α.

$$DN(R^i_k) = DN(R^{i-\alpha}_k)^\alpha_O = SN(R_k)^i_O \quad (2a)$$
$$SN(R_k) = SN(R_{k-\alpha})^\alpha_{O'} = SN(R_0)^k_{O'} \quad (2b)$$

According to Equation (2), at iteration i, the dynamic name of register R_k can be generated through shuffling the corresponding static register name $SN(R_k)$ by an offset of O for i times, while $SN(R_k)$ should be generated through shuffling the static name of the 0th destination register $SN(R_0)$ by an offset of O' for k times. These equations clearly confirm that at each iteration, the dynamic register names can be completely determined by the compiler.

To effectively balance register accesses while minimizing the hardware complexity, a light-weight shuffling function is furthermore necessitated. Given a *shuffle window* composed of a set of contiguous physical registers B, B+1, ..., B+T−1, an effective shuffling function needs to ensure that each logical destination register R_k will be sequentially mapped to each physical register within the window before it shuffles back to the initial mapping. In other words, the shuffling function needs to establish a one-to-one mapping from V^1_O, V^2_O, ..., V^T_O, the T consecutive dynamic names of register R_k, to the T distinct values B, B+1, ..., B+T−1, within the shuffle window. According to this requirement, **two** shuffle functions, namely, a modulo addition and a *Galois field* multiplication, can be employed to attain a deterministic register shifting, as summarized in the following equation.

$$SN(R_k) = B^k_{O'} = \begin{cases} B \otimes 2^{T-(k*O)\%T}, & T = 2^n - 1; \\ (B - k*O)\%T, & \text{GCD}(O, T) = 1. \end{cases} \quad (3)$$

In modulo T addition, the static register name of R_k is generated as $(B - k*O)\%T$. The values of the offset O and the window size T should be relatively *prime* so as to ensure the assignment of distinct physical names to different logic registers within a single iteration. In hardware, this addition function can be implemented using a j-bit modulo T adder for each register access port, assuming a total of 2^j registers provided in the architecture. On the other hand, if the value of the window size T equals $2^n - 1$, a more efficient shuffle function can be implemented so that $B^k_{O'}$= $B \otimes 2^{T-(i*O)\%T}$, with $\otimes$ denoting the multiplication operators defined in the extension *Galois field* of $GF(2^n)$. The hardware implementation of this multiplication function is comparatively cheaper as no modulo adders but only a limited number of *xor* gates are required.

The differences between these two functions are concretely illustrated in Table I, which shows the mapping of the five destination registers of the *bzip2* example in 7 consecutive iterations, with B, O and T set to 1, 1, and 7, respectively.

C. Loop-carried dependence preservation

As the new name of each logical destination register can be determined using Equation (3), the names of source registers

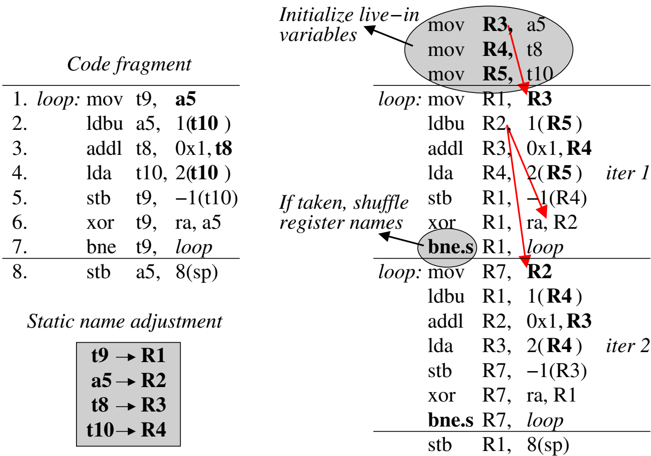

Fig. 3. Register name adjustment in two consecutive iterations

can be determined accordingly. Since the mapping of a logical register varies across iterations, read accesses before and after the first assignment within the loop body should be directed to distinct physical registers. Specifically, all the read accesses following the first write operation, as it remaps the logical register, should be directed to the new allocated physical register. In contrast, all the read accesses preceding the first write operation should obtain the value produced at the prior iteration, thus requiring the compiler to additionally shuffle the register name by O'.

The aforementioned name adjustment of source registers can be illustrated more clearly by considering the logical register a_5 in the *bzip2* loop presented in Figure 2a. As shown in the first column of Table I, the name of a_5 is adjusted to $B^1_{O'}$ by the compiler. Since a_5 is remapped by instruction 2, all the subsequent read accesses to a_5 within the loop body should be directed to $B^1_{O'}$. On the other hand, instruction 1, which reads a_5 before it is remapped, should obtain the value produced at the prior iteration wherein the name of a_5 is not $B^1_{O'}$ but $B^2_{O'}$. Accordingly, the compiler should adjust the name of a_5 appearing in instruction 1 by an additional amount of O' so as to preserve this loop-carried dependence.

An additional shuffle of O' to the names of the *live-in* variables allows register values to be effectively passed across loop boundaries during execution. Therefore, semantic correctness can be naturally guaranteed as long as *live-in* variables, such as a_5, t_8 and t_{10} in the *bzip2* loop, are correctly initialized prior to entering the loop. This task can be attained simply through the insertion of extra *move* instructions to transfer register values prior to entering the loop. These few register *move* instructions, as they are executed quite rarely outside the loop body, introduce no overhead in practice, neither in terms of performance nor in terms of energy.

To concretely illustrate the aforementioned name adjustment policy for destination and source registers, it has been applied to a non-unrolled version of the *bzip2* loop presented in Figure 3. Using the *modulo addition* in Table I as the shuffle function, Figure 3 presents the register names in the first two iterations of the transformed code. As can be seen, the compiler has globally adjusted register names according to the order in which

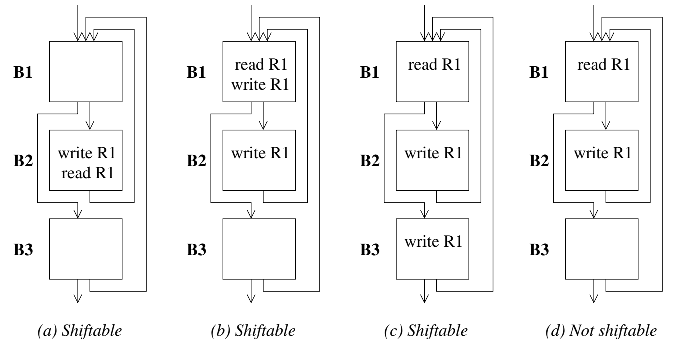

Fig. 4. Shiftability analysis of register *R1*

they appear as destinations. The names of adjacent yet distinct register destinations differ by an offset of O'=-1, while an extra offset of O' is added to each live-in read reference shown in instructions 1, 2, 3, and 4. Meanwhile, a hint is inserted into instruction 7, the loop branch, so that once the branch is taken, each register name, except for the read-only register *ra*, will be shifted by an additional offset of O=1. Finally, three register *mov* instructions have been inserted prior to entering the loop so as to initialize the live-in registers a_5, t_8, and t_{10}, respectively.

D. Shiftable logical register identification

The proposed register shuffling scheme requires a detailed examination of register access patterns so as to determine whether a logical register accessed within the loop body is *shiftable* or not. In general, the characteristics of the proposed register shuffling scheme preclude its application to **two** types of logical registers. Firstly, as a logical register is remapped upon the first assignment, *read-only* registers, such as *ra* in the *bzip2* example, become unshiftable. A more complex case is that of registers exhibiting conditional definitions within the loop body; as the compiler needs to identify for each read access the exact iteration at which the value is produced, a logical register cannot be shuffled if its value is not certifiably updated at each loop iteration, that is, if it exhibits *write* accesses only in *conditionally executed basic blocks* but *read* accesses outside those blocks.

Conditionally defined registers create an issue of **nondeterministic** loop-carried dependences, which can be illustrated more clearly through examining the four cases presented in Figure 4. These four cases share the same control flow yet exhibit a variety of access patterns to register *R1* within the loop body. In Figure 4a, the write access in basic block *B2* constitutes a conditional definition. However, *R1* is still *shiftable* since it is read within the same basic block following such a write access, thus allowing identical register names to be assigned to both accesses. In Figure 4b, *R1* is also shiftable as the write access in *B1* constitutes an unconditional definition, thus indicating that the read access in *B1* should always obtain the value defined in the preceding iteration. Similarly, in Figure 4c, while neither of the write accesses in *B2* and *B3* is guaranteed to be executed, the two accesses in conjunction constitute an unconditional definition, thus making *R1* shiftable. In comparison, in Figure 4d, *R1* is only written on the fall-through path of the branch, resulting in the read access

TABLE II
ACCESS PATTERN-BASED REGISTER CLASSIFICATION

Shiftable	Unconditionally *written* Conditionally *written* and *read* on the same path
Unshiftable	*Read*-only Conditionally *written* yet unconditionally *read*
Free	*Not-accessed*, either *dead* or *live*

in *B1* obtaining a value defined in either the preceding iteration or an even earlier iteration, depending on the branch outcome. As a result, for such a read access in *B1*, the compiler cannot statically determine the exact iteration at which the value is produced, resulting in *R1* being *unshiftable*.

An *unshiftable* logical register does not need to be remapped, if it is not accessed frequently within the loop body. However, in the extreme case of an unshiftable register being frequently accessed, **two** approaches can be adopted to prevent local heat buildup. In a hardware-oriented approach, the value of such a register can be duplicated into a dedicated buffer for access, instead of the power-hungry register file. In a software-oriented approach, an extra *move* instruction can be inserted within the loop body to make it *shiftable*. If this register happens to be a conditionally defined register (for example, *R1* in Figure 4d), such a *move* instruction can be inserted into the basic block executed on the other path of the branch (*B3* in Figure 4d). If, on the other hand, the frequently accessed yet unshiftable register happens to be a read-only register, the extra *move* instruction needs to be inserted into an unconditionally executed basic block.

E. Physical register reallocability analysis

The example presented in Section IV-A indicates that the proposed deterministic shuffling approach requires the existence of at least **one** free extra register, such as t_{12} in Figure 2c, for the shuffle of the first destination within the loop body. As most execution hotspots are composed of nested loops consisting of only a limited number of instructions, the requirement of one free register can be naturally satisfied since typically only a subset of registers is accessed during loop execution. The *bzip2* example presented in Figure 2a clearly confirms this property in that only 6 out of the total 32 registers are accessed within the loop body.

While theoretically the shuffle window only needs to include one extra free register in addition to the shiftable destination registers, the search for an increasingly balanced register access distribution motivates the maximization of the number of free registers within the shuffle window. A detailed examination indicates that according to the access pattern, all the logical registers and hence, the corresponding physical registers, can be classified into three categories: *shiftable*, *unshiftable*, and *not-accessed*. For the third type, a physical register not accessed within the loop body can be directly remapped, if it is not used to hold a live variable with infinite lifetime across the execution of the whole loop. As an example, in the *bzip2* loop all the *not-accessed* registers except for *sp* are free for remapping. Register *sp*, on the other hand, holds its lifetime across the whole loop as it is directly read after exiting the loop. However, even this type of *not-accessed* yet *live* registers can be freed up through

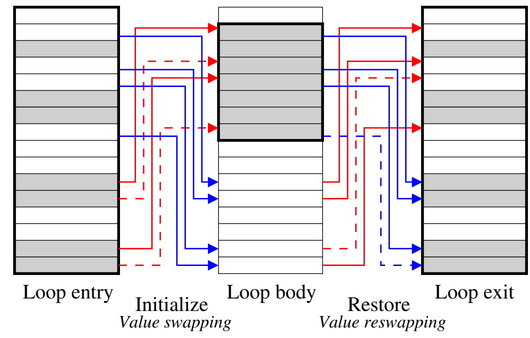

Fig. 5. Building a shuffle window through swapping register values at loop entry and exit

employing extra store and load instructions to checkpoint and restore the original value at loop entries and exits, respectively. The introduced performance overhead is practically nonexistent since this checkpointing and restoration process is performed outside the loop execution.

By checkpointing and restoring *live* yet *not-accessed* register values, all the registers that are not accessed within the loop body become available for remapping. Accordingly, among the three classes of registers listed in Table II, both the *shiftable* and the *free* registers are included in the shuffle window, while only the *unshiftable* registers need to be placed outside the shuffle window. The size of the shuffle window therefore can be maximized, thus enabling the achievement of a more balanced access distribution and, hence, a further reduction in peak temperature.

The identified shiftable and free registers may *scatter* across the entire register file. As the shuffle window should be composed of a set of *contiguous* registers, at the entry and the exit of each frequently executed loop, some register values need to be swapped so that the identified *shiftable* and *free* registers can be placed at contiguous positions. This process is concretely presented in Figure 5. At the loop entry the *live-in* register values need to be preserved, implying that *unshiftable* registers within the shuffle window need to be swapped out, while *shiftable* yet *live-in* registers outside the window need to be swapped in. On the other hand, at the loop exit, a register re-swapping process needs to be performed to preserve the *live-out* register values, both within and outside the shuffle window. Both the register swapping and reswapping processes are accomplished by the compiler through the insertion of extra *move* instructions which, as they are executed outside the loop body, introduce no overhead in practice.

F. Functional Evaluation

We have discussed the proposed deterministic register shuffling technique from three vantage points, namely, the dynamic shuffling functions, the adjustment of logic register names, as well as the identification of the shiftable and free registers. Since the proposed technique only remaps register names across loop iterations, it can be independently applied on each execution hotspot, i.e., a frequently executed loop. Due to the iterative nature and the relatively short static code size of each loop, the proposed technique delivers maximum benefit at minimal cost, as only 10% of the code needs to be analyzed

while balanced register accesses for 90% of execution time can be accomplished.

Compared to the thermal-aware register reassignment approaches [20], [18], the proposed deterministic register shuffling technique requires no revisitation of the NP-hard register allocation problem to perform live range reassignment. Therefore, the adjustment of logic register names can be implemented as a procedure to be performed subsequent to the conventional register allocation phase, thus retaining all the concomitant benefits of the latter. Moreover, a detailed examination indicates that neither of the two techniques can fully balance the accesses to each individual register at each loop iteration. Instead, both techniques attain a relatively *coarse-grained* access balance, yet one exploits the spatial domain while the other exploits the temporal domain. The thermal-aware register reassignment approaches attain a *spatial* balance at the granularity of register sub-banks, thus restricting their applicability solely to multi-bank register files. In contrast, the proposed technique aims to attain a *temporal* balance for each individual register at the granularity of loop iterations. As temperature takes at least 0.1 million cycles to rise by 0.1 $°C$ [6], this iteratively balanced access activity thus enables an effective reduction of peak temperature even for single-bank register files.

As the proposed technique deterministically shuffles register mapping across iterations, the attainable benefits in terms of reliability enhancement are maximized when it is applied to single processor architectures with no explicit register renaming support. For architectures with pure dynamic register renaming, such as conventional superscalar processors, a large hardware mapping table needs to be maintained so as to eliminate pseudo register name dependences. As this mapping table needs to be accessed using logical register names at a frequency no lower than that of register file accesses, it becomes a temperature "hotspot" with skewed access distribution. In this case, the proposed technique can be employed to evenly distribute the accesses to different entries within the mapping table.

Additionally, future computer systems are expected to intensively use multicore architectures, for which thermal induced reliability aspects have already been identified as a grand challenge. As such systems typically scale upwards in the number of cores but not necessarily in the complexity of each core, the proposed technique despite the possible absence of the renaming logic, can be employed to effectively reduce the register file peak temperature for each core and hence, to improve the reliability of the entire system.

V. IMPLEMENTATION

The implementation of the proposed deterministic register shuffling technique consists of two collaborative parts, a compilation procedure that embeds regularity into static register names, as well as a hardware implementation of a shuffling function that dynamically determines the name of a register at each iteration.

A. Static register name adjustment

The pseudo-code for adjusting logic register names is outlined in Algorithm 1. This procedure only relies on the profiling information regarding the *execution counts* of each basic block, based on which a set of functions have been developed to accomplish static register name adjustment. Specifically, each frequently executed loop is transformed in the following 5 steps:

Algorithm 1 Register Name Adjustment

```
1: for each procedure do
2:    for each frequently executed loop do
3:       Differentiate shiftable and unshiftable registers;
4:       Calculate AveAccessCnt;
5:       if AccessCnt(R_j) > AveAccessCnt for a unshiftable R_j
         then
6:          Insert an extra mov to make R_j shiftable;
7:       end if
8:       Insert extra store and load to free up not-accessed yet live
         registers;
9:       T = N_total − N_unshiftable, and select B and O thereafter;
10:      Order the shiftable destination registers;
11:      Globally adjust register names such that the static name of
         the k^th register SN(R_k) = B^k_O';
12:      Shuffle the name of each live-in variable by an extra offset
         O';
13:      Insert a hint in the loop branch;
14:      Insert extra mov to initialize live-in variables at loop entry
         and restore live-out variables at loop exit;
15:   end for
16:   Globally perform register coalescing outside the renamed
      loops to eliminate redundant mov instructions;
17: end for
```

- Partition shiftable and unshiftable registers (lines 3-7);
- Free up *not-accessed yet live* registers (line 8);
- Determine shuffle functions (line 9);
- Sequentially adjust names of destination and source registers (lines 10-13);
- Initialize *live-in* variables and restore *live-out* variables at loop entry and loop exit, respectively (line 14);

As the goal of the register shuffling technique is to preclude local heat buildup through iterative mapping of a hot logical register to distinct physical registers, the algorithm inserts extra *move* instructions to shuffle a frequently accessed register (line 6), if it is detected to be unshiftable (line 5). These *move* instructions, together with the *store* and *load* instructions inserted for freeing up *not-accessed yet live* registers (line 8) and the *move* instructions inserted for *live-in* or *live-out* variables (line 14), constitute the overhead of the proposed technique. As most of these extra instructions are executed outside the loop body, the overhead in execution time is negligible. Such overhead can be further reduced through performing an extra step of register coalescing [22] on the transformed code (line 16) so as to eliminate redundant move instructions.

B. Dynamic register name shuffling

Using the code transformation support outlined in Algorithm 1, a deterministic register shuffling process can be accomplished during execution, as long as the hardware is informed by the compiler about the *shuffle vector*, <B, O, T>, prior to entering a frequently executed loop.

Using the $GF(2^3)$ *multiplication* in Table I as the shuffle function wherein the vector <B, O, T> = <1, 1, 7>, the circuit presented in Figure 6 can be employed to convert

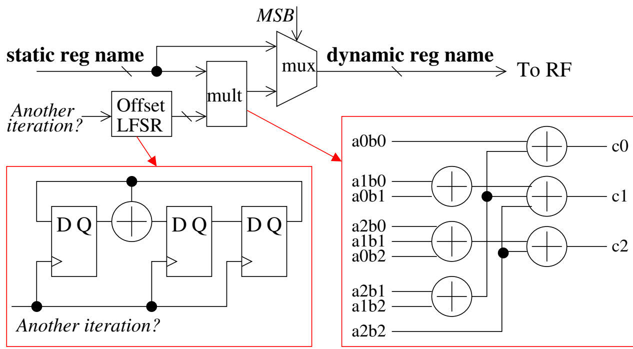

Fig. 6. Gate-level logic for translating register names

logical register names to physical register indices for the *bzip2* example. As can be seen, during loop execution each logical register name is multiplied by the value of the offset register. The offset value is multiplied by 2 whenever a loop branch is encountered, implemented through shifting the 3-bit LFSR one bit to the right. Meanwhile, as in this example the shuffle window is composed of registers from R1 to R7, the most significant bits of the static encoded register name are used to differentiate whether the register falls within the shuffle window. If it is, the register is shiftable, resulting in the use of the multiplier's result as the physical register index. Otherwise, the logical name of the unshiftable register is directly used to access the register file.

It needs to be noted that the implementation shown in Figure 6 corresponds to the example shown in Figure 3. The implementation parameters are for illustrative purposes only, and can be customized according to the register utilization characteristic of the application. More concretely, it can be clearly seen from Figure 6 that the proposed register shuffling technique requires no hardware mapping table but only a n-bit *GF* multiplier and a n-bit 2-to-1 multiplexer for each register access port, together with a single n-bit LFSR to record the shuffling offset. Moreover, with the selection of an appropriate field generating polynomial, GF multipliers can be efficiently implemented using a small set of AND and XOR gates. For 3 to 7 bits parallel field multipliers, the cost-effective polynomial as well as the total gate count and longest path of the corresponding implementation have been listed in Table III.

The *GF multipliers* shown in Table III require a size of $2^n - 1$ registers for the shuffle window. In contrast, the *modulo addition* can be employed more generally as the shuffling function for shuffle windows of other sizes. In this case, the proposed register shuffling technique requires a n-bit modulo adder, a n-bit comparator, and a n-bit 2-to-1 multiplexer for each register access port, together with a single n-bit adder and a n-bit register to calculate and record the shuffling offset. Although this additional necessitated hardware is more complex than the *xor* gate based implementation of the *GF* multipliers, it is still negligibly small compared to the mapping tables used in conventional register renaming techniques.

As both the logic and the physical register names preserve all the true data dependences within the loop body, the behavior of the rest of the pipeline, such as the forwarding logic, would not be affected by the register shuffling process. Moreover, since register write accesses are typically performed at a later pipeline stage, the translation of register names can be performed in parallel with the calculation of the instruction result. Even for register read accesses, the access latency of the small translation logic can also be effectively hidden, since in the typical case cache accesses constitute the longest pipeline stage.

TABLE III
THE DESIGN COMPLEXITY OF GF MULTIPLIERS

Window size	**Field polynomial**	**Gates**	**Longest path**
7 regs	$x^3 + x + 1$	15	1 AND + 2 XOR
15 regs	$x^4 + x + 1$	25	1 AND + 2 XOR
31 regs	$x^5 + x^2 + 1$	37	1 AND + 2 XOR
63 regs	$x^6 + x + 1$	54	1 AND + 3 XOR
127 regs	$x^7 + x + 1$	72	1 AND + 3 XOR

VI. SIMULATION RESULTS

In this section we experimentally evaluate the efficacy of the proposed register rotation technique in balancing register accesses, reducing the chip-wide peak temperature, and improving processor reliability. To evaluate the proposed technique for different types of applications, a set of experimental studies have been performed on both the Mediabench [23] and the SPECint 2000 benchmarks.

A. Register Access Results

The discussions presented in Sections IV-D and IV-E clearly show that the partition of shiftable/unshiftable registers and, hence, the effectiveness of the proposed register shuffling technique are strongly related to register access characteristics. As a result, the first step in our experimental evaluation is the examination, for each loop, of the numbers of *read-only* registers, *conditionally defined* yet *unconditionally referred* registers, and registers *not accessed* in the loop. This is achieved through using ATOM [24] to instrument the assembly code to identify execution hotspots (i.e., frequently executed loops) and to generate register usage profiles. The control flow and register usage information of each loop are analyzed thereafter.

The collected profiling results are presented in Table IV. Only the profiling results for the selected SPEC 2000 benchmarks are presented, since these benchmarks exhibit a more balanced register utilization than the Mediabench [23] programs due to their relatively larger working sets. For each benchmark, we report the number of hot loops that have been identified, their occupancy in the total execution time, as well as six sets of register usage data. Table IV lists the maximal, the average, and the minimal number of *not-accessed* registers and *live not-accessed* registers, as well as the maximal and the average number of *read-only*, *hot read-only*, *cond-defined*, and *hot cond-defined* registers. The minimal values are not listed for the last four sets since these values are always 0.

The results regarding the minimal number of *not-accessed* registers indicate that **all** the hot loops identified by ATOM have at least 1 free register, thus clearly confirming the wide applicability of the proposed register shuffling technique. Due to the small code size, the average number of registers accessed within a loop body is less than 9. This highly skewed register utilization clearly confirms the necessity for register shuffling techniques, such as the one we herein propose, so as to deliver

TABLE IV
THE NUMBER OF HOT LOOPS, THEIR OCCUPANCY IN EXECUTION TIME, AND REGISTER USAGE INFORMATION

Benchmark		art	bzip2	mcf	twolf
Hot loop # Execution time		32 79.8%	60 78.3%	29 64.8%	61 71.9%
Not-accessed #	max aver min	28 25.34 14	29 22.33 13	29 23.66 13	29 22.56 1
Live not-accessed #	max aver min	20 10.47 0	13 5 0	13 3.76 1	19 5.98 0
Read-only #	max aver	6 1.91	6 2.33	7 2.24	7 2.02
Hot read-only #	max aver	4 1	4 1.73	3 1.14	3 1.08
Cond-defined #	max aver	1 0.06	3 0.17	3 0.28	3 0.21
Hot cond-defined #	max aver	1 0.06	1 0.03	1 0.03	3 0.11

a more balanced access distribution. Meanwhile, Table IV also shows that on average a loop contains only 2 *read-only* registers, and only 1 of them needs to be rotated to prevent local heat buildup. The number of *conditionally defined* yet *unconditionally referred* registers is even less, as most hot loops are composed of a limited number of basic blocks. These values clearly confirm that a highly limited number of extra *mov* instructions (less than two on average) would suffice to make this small set of *hot read-only* and *hot conditionally defined* registers shiftable.

According to the register usage profiles generated by ATOM, the new register names are statically determined, based on which the SimpleScalar toolset [25] is modified to implement the proposed register shuffling technique on top of an in-order 2-way processor. We furthermore compare the proposed technique with the thermal-aware register reassignment technique [20]. Assuming that the register file is composed of 8 sub-banks, **two** sets of data are reported, namely, the access distribution to each *individual register* and the access distribution to each *sub-bank*. The cumulative ratios of the most frequently accessed registers and sub-banks are shown in Figure 7.

As can be seen, for the four SPEC2000 benchmarks, the proposed technique can achieve a more balanced access distribution to each individual register as compared to the thermal-aware register reassignment technique [20]. More concretely, initially 81% to 94% of all the register accesses are mapped to 12 registers which, in a completely balanced case, should only capture $12/32 = 37.5\%$ of total accesses. Using static register reassignment (the top-left quadrant in Figure 7), 64% to 80% of total accesses are mapped to 12 registers, while using the proposed register shuffling technique (the top-right quadrant), only 50% to 60% of total accesses are mapped to 12 registers.

If the access distribution is evaluated at the granularity of register sub-banks, both techniques can achieve a quite balanced access distribution in that only 38% to 40% of all the register accesses are mapped to 12 registers. Compared to the reassignment [20] technique, the proposed shuffling technique results, for *mcf* and *twolf*, in a slightly elevated

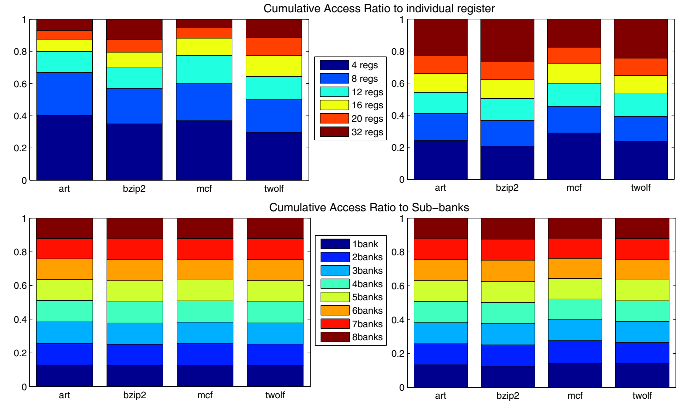

Fig. 7. Reduction in peak temperature of the entire chip

amount of accesses (less than 2%) hitting the first subbank. This is because register R0, which cannot be rotated since its value corresponds strictly to 0, happens to be a frequently accessed register within several loop bodies. In the reassignment [20] technique, R0 can be placed into a subbank with a set of "cold" registers so as to balance the access counts to that subbank. However, in the proposed technique, these "cold" registers are iteratively mapped to "hot" logical registers. The increased amount of accesses thus results in the corresponding subbank being accessed slightly more frequently than the remaining subbanks.

B. Temperature Results

Our next step of evaluation focuses on the generation of temperature profiles. WATTCH [26] is modified to generate energy profiles of each hardware resource, especially each register within the register file. The power consumed by the small 5-bit adder and multiplexer is also included in each register file access. The aggressive clock gating provided by WATTCH is used to avoid unnecessary power consumption. Using these energy profiles, Hotspot [6] is employed to sample the transient temperature of each hardware resource. This sampling interval is set to 20,000 instructions, which is substantially less than the thermal time constant of any hardware resource. An Itanium-like [27] processor shown in Figure 8 is used as the floorplan input to Hotspot. The die size is set to $8mm \times 8mm$, and the initial temperature is set to $60°C$.

The obtained reduction in chip-wide peak temperature is presented in Figure 9. As can be seen, the proposed register

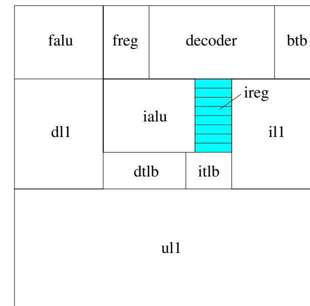

Fig. 8. The processor floorplan used in simulation

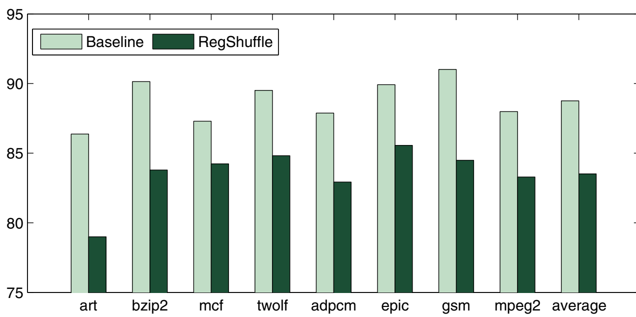

Fig. 9. Reduction in peak temperature of the entire chip

file access balancing technique can achieve a reduction of 3.1 to 7.4°C in chip-wide peak temperature. The highest reduction is achieved in *art*, while the lowest reduction occurs in *mcf*. These temperature results are consistent with the register access results, since a more balanced access distribution is achieved in *art*, as compared to *mcf*.

The simulation results confirm that by targeting the register file, one of the most overheated hardware units in a processor, the proposed technique can effectively reduce the chip-wide peak temperature during program execution. While the amount of temperature reduction seems to be insignificant at first sight, it actually can effectively reduce the fault rate of the entire chip, since the fault rate doubles for every 10°C increase in temperature [12]. Meanwhile, previous studies have shown that a large number of delay violations would occur if the peak temperature exceeds 85°C [4], [6]. It can be seen from the results that for most benchmarks, the proposed algorithm can effectively reduce the peak temperature to below 84°C. On average, the proposed technique achieves a reduction from 88.8°C to 83.5°C.

VII. Conclusions

We have presented in this paper a technique for improving the reliability of an entire chip through reductions in the peak temperature of the register file, one of the most overheated modules in a processor. Peak temperature can be effectively controlled through a register shuffling process that physically remaps the heavily accessed logical registers before heat gets locally accumulated. Furthermore, through the exploitation of application-specific access profiles, the compiler can deterministically control the register shuffling process, thus maximizing peak power reduction within a limited hardware budget and negligible performance degradation. This highly reduced hardware complexity enables the proposed technique to be easily incorporated into most embedded processors so as to effectively reduce peak temperature of the entire chip. Simulation results of SPEC2000 and mediabench programs furthermore confirm that the proposed register shuffling technique can achieve a 1.5 to 3 times more balanced access distribution and a reduction of 3.1 to 7.4°C in chip-wide peak temperature. Such a temperature reduction in turn effectively reduces the amount of run-time faults, thus improving the reliability of the entire chip.

References

[1] R. Blish and N. Durrant, "Semiconductor device reliability failure models," International SEMATECH, Tech. Rep., May 2000.

[2] R. Viswanath, V. Wakharkar, A. Watwe, and V. Lebonheur, "Thermal performance challenges from silicon to systems," Intel Corp., Tech. Rep., 2000.

[3] H. Kufluoglu and M. A. Alam, "A computational model of NBTI and hot carrier injection time-exponents for MOSFET reliability," *Journal of Computational Electronics*, 3(3):165–169, Oct. 2004.

[4] H. Goto, S. Nakamura, and K. Iwasaki, "Experimental fault analysis of 1MB SRAM chips," in *VTS'97*, pp. 31–36, April 1997.

[5] A. H. Ajami, K. Banerjee, and M. Pedram, "Modeling and analysis of nonuniform substrate temperature effects on global ULSI interconnects," *IEEE Trans. on CAD*, 24(6):849–861, June 2005.

[6] K. Skadron, M. Stan, W. Huang, S. Velusamy, K. Sankaranarayanan, and D. Tarjan, "Temperature-aware microarchitecture," in *30th ISCA*, pp. 2–12, June 2003.

[7] A. K. Coskun, T. S. Rosing, K. Whisnant, and K. Gross, "Dynamic temperature-aware scheduling for multiprocessor SoCs," *IEEE Trans. on VLSI*, 16(9):1127–1140, Sept. 2008.

[8] B. C. Schafer, Y. Lee, and T. Kim, "Temperature-aware compilation for VLIW processors," in *13th RTCSA*, pp. 426–431, Aug. 2007.

[9] S. Manne, A. Klauser, and D. Grunwald, "Pipeline gating: Speculation control for energy reduction," in *25th ISCA*, pp. 132–141, June 1998.

[10] K. Skadron, T. Abdelzaher, and M. Stan, "Control-theoretic techniques and thermal-RC modeling for accurate and localized dynamic thermal management," in *HPCA-8*, pp. 17–28, Feb. 2002.

[11] P. E. Dodd, M. R. Shaneyfelt, J. R. Schwank, and G. Hash, "Neutron-induced latchup in SRAMs at ground level," in *41st IRPS*, pp. 51–55, April 2003.

[12] C. J. Lasance, "Thermally driven reliability issues in microelectronic systems: Status-quo and challenges," *Microelectronics Reliability*, 43(12):1969–1974, Dec. 2003.

[13] A. Baniasadi and A. Moshovos, "Instruction flow-based frontend throttling for power-aware high-performance processors," in *ISLPED'01*, pp. 16–21, Aug. 2001.

[14] D. Brooks and M. Martonosi, "Dynamic thermal management for high-performance microprocessors," in *HPCA-7*, pp. 171–182, Jan. 2001.

[15] S. Heo, K. Barr, and K. Asanovic, "Reducing power density through activity migration," in *ISLPED'03*, pp. 217–222, Aug. 2003.

[16] P. Chaparro, J. Gonzalez, and A. Gonzalez, "Thermal-aware clustered microarchitectures," in *ICCD 2004*, pp. 48–53, Oct. 2004.

[17] M. D. Powell, M. Gomaa, and T. N. Vijaykumar, "Heat-and-run: Leveraging SMT and CMP to manage power density through the operating system," in *ASPLOS'04*, pp. 260–270, Oct. 2004.

[18] J.-L. Cruz, A. Gonzalez, M. Valero, and N. P. Topham, "Multiple-banked register file architectures," in *27th ISCA*, June 2000, pp. 316–325.

[19] M. Kondo and H. Nakamura, "A small, fast and low-power register file by bit-partitioning," in *HPCA-11*, pp. 40–49, Jan. 2005.

[20] X. Zhou, C. Yu, and P. Petrov, "Compiler-driven register reassignment for register file power-density and temperature reduction," in *45th DAC*, pp. 750–753, June 2008.

[21] W.-W. Hsieh and T.-T. Hwang, "Thermal-aware post compilation for VLIW architectures," in *ASP-DAC*, pp. 606–611, Jan. 2009.

[22] L. George and A. W. Appel, "Iterated register coalescing," *ACM Trans. on Programming Languages and Systems*, 18(13):300–324, 1996.

[23] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "Mediabench: A tool for evaluating and synthesizing multimedia and communications systems," in *30th Micro*, pp. 330–335, Dec. 1997.

[24] A. Srivastava and A. Eustace, "ATOM: A system for building customized program analysis tools." Western Research Lab, Tech. Rep., 1994.

[25] T. Austin, E. Larson, and D.Ernst, "Simplescalar: an infrastructure for computer system modeling," *Computer*, 35(2):59–67, Feb. 2002.

[26] D. Brooks, V. Tiwari, and M. Martonosi, "WATTCH: A framework for architectural-level power analysis and optimizations," in *27th ISCA*, May 2000, pp. 83–94.

[27] S. Rusu and G. Singer, "The first IA-64 microprocessor," *IEEE Journal of Solid-State Circuits*, 35(11):1539–1544, Nov 2000.