Thermal-Aware Compilation for System-on-Chip Processing Architectures

Mohamed M. Sabry
Embedded Systems
Laboratory, EPFL
EPFL-STI-IEL-ESL
1015 Lausanne, Switzerland
mohamed.sabry@epfl.ch

José L. Ayala
DACYA
Complutense University of
Madrid
28040 Madrid, Spain
jayala@fdi.ucm.es

David Atienza
Embedded Systems
Laboratory, EPFL
EPFL-STI-IEL-ESL
1015 Lausanne, Switzerland
david.atienza@epfl.ch

ABSTRACT

The development of compiler-based mechanisms to reduce the percentage of hotspots and optimize the thermal profile of large register files has become an important issue. Thermal hotspots have been known to cause severe reliability issues, while the thermal profile of the devices is also related to the leakage power consumption and the cooling cost. In this paper we propose several compilation techniques that, based on an efficient register allocation mechanism, reduce the percentage of hotspots in the register file and uniformly distribute the heat. As a result, the thermal profile and reliability of the device is clearly improved. Simulation results show that the proposed flow achieved 91% reduction of hotspots and 11% reduction of the peak temperature.

Categories and Subject Descriptors: D.3.4 Programming Languages:Processors [Compilers]; B.8 Performance and Reliability.

General Terms: Algorithms, Management, Reliability.

Keywords: Thermal-aware, Compiler, Register-file.

1. INTRODUCTION

Temperature dissipation is an important factor in the performance and reliability of embedded systems. With the advent of new technologies and scaling design parameters, thermal issues have emerged as one of the key design parameters that need to be addressed.

Thermal dissipation in integrated circuits has a negative effect on multiple aspects. On one hand, leakage current (one of the main sources of power dissipation in todays sub-micron technologies) presents an exponential dependency with temperature [1]. Secondly, temperature has a direct impact on the reliability of the system, because several processes are driven by the increase of temperature or the spatial and temporal gradients that appear during normal functioning. Temperatures over a threshold in localized areas of the chip (hotspots) can produce timing delay variations, transient reduction in overall system performance, or even permanent damages in the devices [2]. Moreover, the reliability factors do not only depend on the average temperature of the chip, but the spatial and temporal variations have a strong influence in phenomena like electro migration, negative bias temperature instability, or thermal cycles [3]. It has been shown how the Mean Time Between Failure (MTBF) of an IC is divided by 10 for every 300^oC rise in the junction temperature [4]. Finally, system performance is strongly determined by the temperature. With increasing temperature, phonon concentration increases and causes increased scattering. Thus the carrier mobility due to lattice scattering decreases.

These facts explain the strong efforts that nowadays are being done in the area of thermal optimization in electronic circuits. Some of these efforts look for expensive heat dissipater and sinks that improve the thermal dissipation but increase the cost per chip by more than \$1/W [5]. Some other research works are being conducted to tackle the thermal problem at different levels of abstraction. Computer architects develop thermal efficient processor architectures that optimize the thermal behavior by proposing smart ways of sharing the computer resources [6]. Also, temperature depends on the placement of the units in the chip. Placing heavy power consuming units close together will intuitively generate an even higher temperature area in the chip as temperature is additive in nature. In contrast, placing power consuming units close to units that have a moderate power consumption will allow the heat generated to dissipate through these units. Therefore, thermal-aware floorplanning is an intense area of research [7]. Finally, the software part can control the thermal profile of many processor-based systems by the careful execution order of tasks, the assignment of resources, and the code generation phase. In this area, compilers can play an important role.

Due to its high utilization and relatively small area, the register file has been shown to have the highest peak temperature in several studies [8]. Reducing the register file power density spots would lead to reduction in peak temperatures for both the entire chip and the register file, which in turn would result in improved reliability and reduced leakage power.

The thermal response of the register file is clearly determined by the assignment of registers to the variables defined in the source code, as well as by the profile of accesses to this device. Both parameters can be controlled by the compiler from a software perspective and will lead to the definition of our optimization policies. These techniques should be conceived with a minimal impact on code size and execution time of the application.

This paper proposes a thermal-aware compilation flow that is embedded in a state-of-the-art compiler. The proposed flow introduces a thermal-aware compiler register realloca-

tion based on application-specific information regarding register accesses and frequency of execution, as well as the control flow graph (CFG) of such application. This technique is shown to be an effective stand-alone mechanism for temperature optimization in the microprocessor architecture.

The main contributions of this paper are the followings:

- Analysis of the thermal effects that current register allocators and performance-oriented compiler optimizations have on the register file of register window-based architectures.

- Development of compiler techniques that improve the thermal profile of the register file in terms of mean and peak temperature, as well as percentage of hotspots.

- Integration of the proposed mechanisms in the CoSy compilation flow [9], a retargetable compiler for the generation of high-quality compiled code.

Simulation results show a significant enhancements in terms of reduction of hotspots by 73% on average, as will be shown in section 4.1.

2. RELATED WORK

In the last years, there has been an intense work at the compiler level in power-aware scheduling for VLIW processors, that propose to turn off those unused units to save leakage power [10, 11, 12]. Some of these works [13] explicitly target the register file for energy saving. However, these approaches do not consider temperature as the metric to be optimized and, therefore, the thermal profile is not optimal. Some of the first static approaches to thermal optimization are found in [14, 15], where load balancing heuristics and high-level synthesis techniques are considered.

In Narayanan et al. [16], several techniques are proposed to minimize the thermal emergencies in NoC-based systems through compiler-directed power density reduction. Also, several thermal managing techniques for multicore architectures are explored in Donald and Martonosi [17], and Patel et al. [18] where register temperature reduction through register file and register duplication are investigated.

As ours, a very recent the work by Zhou et al. [19] proposes a register reallocation algorithm for power-density minimization in the register file. However, they target a few specific cases (high-power density registers) in VLIW architectures. VLIW architectures have been also considered in [20], where a thermal-aware instruction generation algorithm is proposed.

Our proposed work differs from previous static approaches in: first, the minimization of several thermal- and reliability-related metrics like the mean and peak temperature of the register file, as well as the percentage of hotspots, with a negligible penalty; second, the development of techniques that cope with the limitations exhibited by register files with register windows; and finally, the integration in a high-quality industrial compilation flow.

3. THERMAL-AWARE REGISTER ASSIGNMENT FLOW

Registers in register file can exhibit a high thermal profile due to:

1. High frequency of accesses (self-heating effect).

2. Proximity of hot registers (mutual diffusion effect).

Based on the thermal profiles collected for different benchmarks, the following observations were noticed:

1. The thermal response of a register will reach a steady state if the frequency of access to such register exceeds a certain threshold, provided the adjacent registers are not allocated. This observation implies that it is more thermal efficient to assign the same register to two variables (iff the frequency of accesses to one of these two variables is above the mentioned threshold) than to allocate two different registers.

2. The thermal profile of the device is improved when the registers are assigned from spread spots of the register file, reducing in this way the mutual diffusion effect.

Based on these observations, register reallocation policies should be designed in order to minimize the number of assigned registers, as well as to locate physically nonadjacent registers. However, if the application exhibits a high register pressure, both constraints could not be jointly satisfied.

Thus, a new thermal-aware compilation flow is proposed to minimize the high thermal profile of the register file. Such flow is designed to minimize the mutual diffusion, as well as self-heating effect. The proposed flow is targeted for various processing architectures but, it mainly targets register window-based or register bank-based processing architectures.

As shown in Figure 3, the proposed flow is integrated with the whole compilation process before the code generation or code emission process, where the pseudo-code generated in the previous phases is translated to target code. This flow is divided into two stages that appear only in register window-based architectures: *Multi-window context switching* and *BBCS*. Then, a third stage that can be applied on any processing architecture, *DIST_MAX*, takes part.
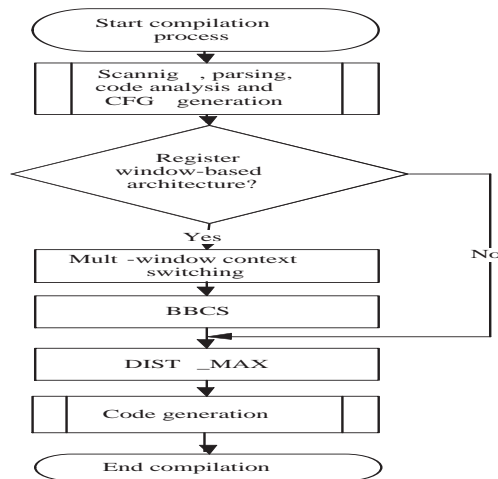

Figure 1: Proposed thermal-aware compilation flow.

Each one of these techniques is deployed to minimize the thermal profile of the register file, but with a different perspective, as will be shown in the following subsections.

3.1 Multi-window context switching

The *Multi-window context switching* technique aims to reduce the mutual thermal diffusion between two adjacent windows allocated due to functional (or sub-functional) calls. For example, assuming a function $F1$ is composed of two main

loops, the first loop contains a call to another function $F2$ while the other loop contains no functional calls. In execution, register window i will be allocated to $F1$. However, since the first loop contains a functional call to $F2$, $F2$ will be executed in the adjacent window $i - 1$. This will lead to the usage of two adjacent register windows that will have a thermal diffusion impact on each other. Thus, the overall thermal profile of the register file will be worsened.

The *multi-window context switching* technique is proposed such that each called function will be executed in a nonadjacent to the recently used window, hence the thermal diffusion between register windows will be diminished in such scenario. The proposed technique shifts from the working register window i to a new one; k, in case of a functional call.

For a register file having N register windows, k is calculated from Equation 1

$$k = (i - 3 + (N\%2) + N)\%N \tag{1}$$

This new window reallocation allows the called function to use window $i - 3$ in case of an even number of register windows within the register file, and window $i - 2$ in case of an odd number. These specific windows are selected since these are the first windows that comes in normal sequence after the adjacent window; $i - 1$. However, if $i - 2$ is chosen for a register file with an even number of register windows, only half of the register file will be utilized. Besides that, the selection of different value for the next window instead of the chosen values might have larger overhead impact and slightly similar performance outcome.

This technique will ameliorate the sequence in which register windows are deployed such that the spatial distance between two consecutively used windows will be increased, as well as the temporal separation between two physically adjacent windows. For example, the sequence on the access to the register windows of a register file with 8 windows would be $0 - 5 - 2 - 7 - 4 - 1 - 6 - 3 - 0$. while the sequence for a register file with 9 windows would be $0 - 7 - 5 - 3 - 1 - 8 - 6 - 4 - 2 - 0$.

The enhancements resulted from applying this technique have an overhead cost, since additional instructions will be needed for such movement. The available instructions [1] that shift the register window can only manage one single window per instruction. Therefore, to shift more than one window, it is required to repeat the execution of the shifting instructions more than once. However, this overhead is found to be negligible, as it will be shown in section 4.1.

3.2 Basic-Block Code Splitter (BBCS)

Basic-Block Code Splitter (or *BBCS*) aims to reduce the self heating effect of a register window by allocating more than a single register window to the same function, regardless the existence of sub-functional calls in such function. This technique will allow a procedure to use two register windows i and $i-1$ instead of just one window. However, these windows are used sequentially not simultaneously (i.e. a portion of the procedure will be executed using register window i, and the rest will be executed using $i - 1$).

This technique explores the whole procedure via its control flow graph (CFG). From the entry basic block, the graph is being explored in a breadth first fashion. For each block, the predecessor and the successor blocks are identified and stored in different lists; *predecessor_list* and *successor_list*.

Each block in the predecessor blocks is checked to be in *predecessor_list*. If it is not, such block index is inserted in

---

[1] in SPARC-like architectures

another list; *notfound_list*. After finishing the processing of predecessor blocks, the current block index is inserted in *predecessor_list*.

After that, each block in the successor blocks is checked to be in *notfound_list*. If it is found, its index is removed from that list. If it is not, such block index is inserted in *successor_list*. The splitting condition is fulfilled when there are no block indices in *notfound_list* and there is only one block index in *successor_list*. When splitting occurs, a microcode is injected to move the live registers to the new window, in addition to the context switching instruction.

Such condition could be elaborated as follows: the blocks executed before the splitting should be dead (i.e. they will not be executed again) by the time the splitting point is reached. This condition could also be rephrased as follows: in order to make a successful splitting, all the nodes in the control flow graph (CFG) should lead into the same basic block BB with no dependency on a block that will be executed after BB.

When the splitting condition is satisfied, the compiler counts the number of input live registers that should be available at the new window, named N_{liveR}. If N_{liveR} is lower than a certain threshold (TH), then the splitting occurs. If not, the algorithm will continue looking for another splitting point. The mentioned threshold is related to the number of *output* registers of the register file; N_{OR}, and the remaining number instructions after the potential splitting block; N_{iB}. TH can be calculated using Equation 2.

$$TH = \begin{cases} 0.05N_{iB} & when \quad 0.05N_{iB} \leq N_{OR} \\ N_{OR} & when \quad otherwise \end{cases} \tag{2}$$

This equation can be interpreted as follows: the instruction overhead resulting from moving the live registers from the old window to the new one should not exceed 10% the number of proceeding instructions until the end of the procedure; N_{iB}. Such limit of instruction overhead is program independent and assumed with this value to diminish the overhead introduced due to context switching from both thermal and code size point of view. Moreover, the instruction overhead is limited by the available number of *output* registers in the window N_{OR}, since it is architecture based limitation and it will not be efficient to use the memory to move the live registers to the new window.

The overhead resulting from moving one live register is 2 instructions; one instruction is required for moving the live register to output register, while an other will be executed after switching to move the input register to its proper location. Therefore, for $N \leq N_{OR}$ registers, the overhead OV equals:

$$OV = 2N \leq 2N_{OR} \tag{3}$$

And since this overhead cannot exceed 10% of the remaining instructions, therefore:

$$OV \leq 0.1N_{iB} \tag{4}$$

By substituting the values of 3 in 4:

$$2N \leq 0.1N_{iB} \tag{5}$$
$$Therefore \ N \leq 0.05N_{iB} \tag{6}$$
$$And \ since \ N \leq N_{OR} \tag{7}$$
$$Therefore \ N \leq TH \tag{8}$$

Where TH is the value defined in Equation 2. For example, if there are 7 output registers and the remaining number of

instructions is greater than 140, then the threshold is 7. However, if the number of remaining instructions is less than 100, then the threshold is 5.

3.3 DIST_MAX

DIST_MAX aims to reduce the thermal diffusion effect between registers within the same register window in a register window-based architecture, or any register file in various architectures. This technique groups the registers in several classification classes. The registers used in a function or a procedure are classified into these classes based on an estimation of the number of accesses to such registers. They are classified as *heavy use, medium use, low use, zeros*, and *system. Zeros* class includes the registers with zero number of access (i.e. unused registers), while *system* class contains registers used by the system that cannot be reallocated such as, the stack pointer and the registers used in passing parameters.

Statistics of registers accesses are evaluated by the compiler, and it is used to classify registers based on standard deviation analysis. Assuming that K number of registers are used, and each register R has a number of accesses N_R. The mean of the number of accesses M_R is defined as:

$$M_R = \frac{\sum_{i=1}^{K} N_{R_i}}{K} \tag{9}$$

And the standard deviation of the number of accesses σ_R, can be calculated as follows:

$$\sigma_R = \sqrt{\frac{\sum_{i=1}^{K} (N_{R_i} - M_R)^2}{K}} \tag{10}$$

Using these values, the registers are classified into the mentioned groups where:

1. *Heavy used* registers have values of number of accesses $\geq M_R + \frac{\sigma_R}{2}$.

2. *Medium used* registers have values of number of accesses $\geq M_R - \frac{\sigma_R}{2}$ and $< M_R + \frac{\sigma_R}{2}$.

3. *Low used* registers have values of number of accesses $< M_R - \frac{\sigma_R}{2}$

After the classification of the registers, DIST_MAX reallocates the registers as follows:

1. *Heavy used* registers are placed at a maximum distance between each other.

2. Each one of these registers is surrounded by zero or *low used* registers.

3. *Medium used* registers are reallocated as a second or third surrounding layer to the *heavy used* ones.

4. The remaining *zero* and *low used* registers are placed in the remaining locations that have not been reallocated.

It is clear that this reallocation mechanism has a direct dependency on the physical layout of the register file. This means that the same number of registers used in a program might have different reallocation maps depending on the layout of the register file. Generally, the layout of the register file could be viewed as a 2D mesh [21]. With this assumption, the term *maximum distance* could be achieved with numerous mapping techniques. In this paper, a simple (yet effective) method has been applied, where a register is identified by the row and column it belongs to. First, *heavy used* registers are

reallocated such that the targeted locations are identified by different rows and columns. Then, the reallocation is continued as mentioned before. This method would reallocate the registers from their preallocated positions in Figure 2(a) to the new positions, as shown in Figure 2(b). This distribution guarantees that each row would have at most a single *heavy used* register, but the number of *medium used, low used*, and *zero* are dependent on the number of used registers in each program.
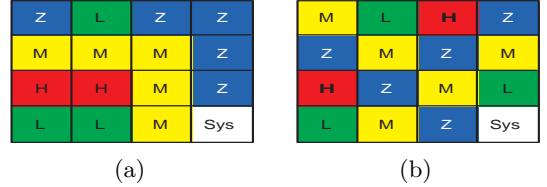

Figure 2: Register reallocation map using DIST_MAX.

DIST_MAX could reallocate any register provided there are no restrictions to the access of such register. However, there are some registers that cannot be reallocated. For instance, the *sys* register in Figure 2(a) is a system register (similar to a stack pointer) and cannot be reallocated to any other location, because it could affect the system execution as well as the probability of misuse of memory contents by the use of the wrong stack address.

4. CASE STUDY: SPARC V8

SPARC V8 architecture has been selected as one of the register window-based architectures [22] (other examples of such architecture are AMP 29k and Intel i960). SPARC V8 is a 32-bit RISC machine with different integer and floating point register files. Register windows are found only in the integer register file, while the floating point register file is a single window, 32 registers register file.

In the SPARC architecture, the instruction format only allows the assignment of registers within the same window. i.e. within the same instruction, the source(s) and the destination registers should belong to the same register window. Moreover, the registers within a single window are classified into *global, output, local,* and *input* registers. The *input* and *output* registers are used for passing and returning parameters in case of a functional call. Thus, these registers cannot be reallocated because, in case of a functional call, wrong parameters could be passed/returned.

In those cases, DIST_MAX has a very small chance of getting a major enhancement in the thermal profile. This is illustrated in Figure 3 that shows the location of the registers within a register file assuming a 2D layout with 8 registers per row. Also, the specifications of SPARC V8 [22] show that there are many limitations in the reallocation of the registers. Even within the same window, various constraints do not work in favor of the reassignment of registers:

1. The stack pointer ($O6$) and the frame pointer ($i6$) cannot be reallocated to another registers.

2. Register $i7$ is used to save the return address for a called procedure.

3. Register $o7$ contains the address of the calling instruction.

4. The *output* registers used for passing parameters cannot be reallocated (variates from a single register to all the existing 5 output registers).
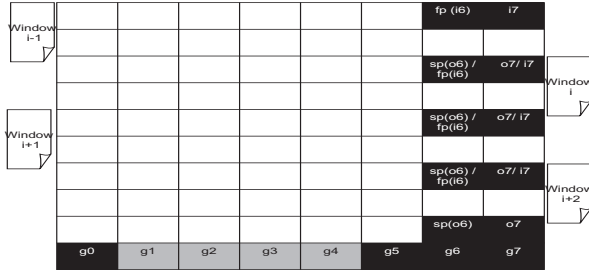
Figure 3: Schematic diagram showing the register file of the SPARC V8. Black registers are irreplaceable, gray registers are replaceable within the same group.

Figure 4: Rate change of hotspots in execution of MPEG2 benchmarks using various compilation techniques.

5. The *input* registers containing the incoming parameters cannot be reallocated.

6. The *global* registers contain global variables used within more than one procedure. Thus, they cannot be interchanged with local, input, or output registers.

These constraints limits the beneficial capability of $DIST\_MAX$ on reallocating the registers within the same window. Fortunately, *multi-window context switching* and *BBCS* are not affected by the mentioned constraints. Hence, the percentage of hotspots and peak temperature have been managed to be reduced, as will be shown in simulation results.

4.1 Simulation results

The experimental work conducted in this work has been performed using the HW-SW emulation platform presented in [23]. This platform is required to extract the power traces corresponding to the execution of the application. This emulation environment allows to implement the core of the SPARC architecture and extract the required thermal statistics, like the profile of accesses to the register file.

The proposed compilation techniques have been embedded in the professional CoSy compilation framework provided by ACE [9]. All the results have been acquired assuming a threshold of 51℃ for hotspots.

The deployed SPARC processor contained an 8 window register file that contains completely 136 registers; 8 *global* and 16 register per window [22]. Benchmarks from MediaBench [24] suite have been applied to measure the proposed flow performance.

Figure 4 shows the rate change of hotspots raised during the execution of the MPEG2 decoding benchmark. This figure shows the execution using the default compilation flow, the combined *BBCS* and $DIST\_MAX$, *multi-window context switching*, and all the proposed compilation flow. Although combining all the techniques reduced the rate of hotspots significantly, there is no significant improvement when using each technique separately. This can be explained by the limitations of the window-based register file, as previously discussed. The *multi-window context switching* spread the usage of windows, but it did not modify the behavior of the register file access within a single window. However, it allowed *BBCS* to make use of multiple windows for the same procedure. Thus, the overall number of access of each register is diminished, which results in a significant reduction of the hotspots.

Besides the reduction of hotspots, the proposed flow also succeeded on the balance of the thermal profile of the register file by reducing the thermal gradients. Figure 5 shows the rate
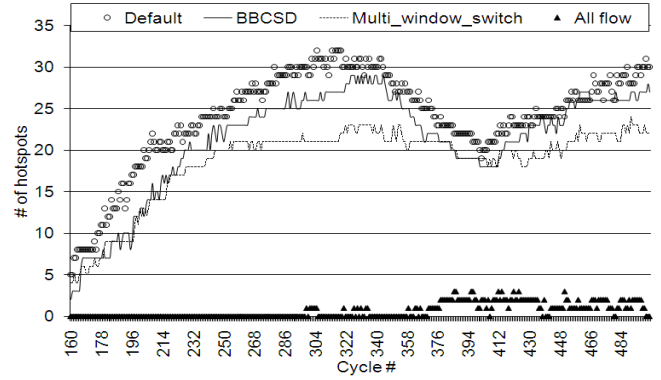
change of the thermal gradient computed as the difference between the maximum and minimum temperatures found on the chip surface per unit area. This figure shows that the thermal gradient is lowered by 38% which means that the variation of temperature within the register file is reduced. This observation along with the reduction of hotspots, implies a more uniform distribution of temperature within the register file.
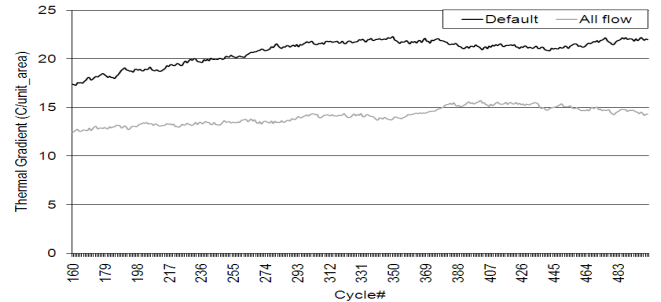

Figure 5: Rate change of thermal gradient of the register file with MPEG.

The proposed compilation flow achieves a significant reduction in both the percentage of hotspots and the peak temperature, as shown in Figures 6 and 7, respectively. It can be noticed that, although the peak temperature of both G711_encode and G711_decode is reduced to values similar to those exhibited by the other applications, the percentage of hotspots is not very much diminished. This indicates that the hotspots were found in a single window of the register file, and that there was not an appreciable impact of neighboring windows. On average, the percentage of hotspots is reduced by 73% with respect to such values of the original compilation, as well as the peak temperature is reduced by 8% with respect to the original peak temperature. The maximum reduction in percentage of hotspots and peak temperature reached 91% and 11%, respectively with MPEG benchmark.

The proposed techniques have a small impact on the code size, which was analyzed in the experimental setup. Table 1 shows the code size of the benchmarks used and the increase due to the extra instructions included by the proposed compilation techniques. The increase in the code size, as seen in the table, can be considered negligible because it did not exceed 0.2%. Since these instructions do not access the memory, there is not any overhead in the dynamic memory size. It is also worth noticing that proposed technique introduced a
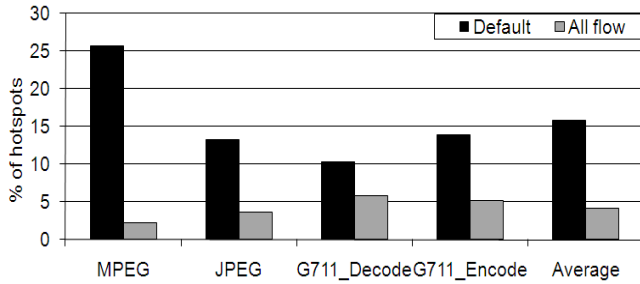
Figure 6: Percentage of hotspots on different applications using the original and modified compilers.
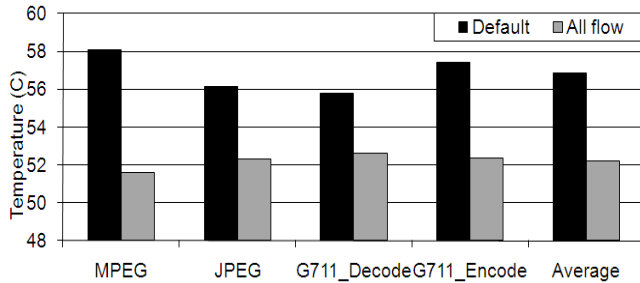

Figure 7: Peak temperature values of different applications using the original and modified compilers.

6% penalty (on average) on compile time, which is negligible as well. However, even if the compile time has experienced a larger penalty, its effectiveness will be negligible when execution time is combined with compile time. Compile time has been measured using the Linux command; *time*.

Table 1: Code size change raised from the proposed compilation techniques.

Benchmark	Original size	New size	% increase
MPEG	435185	435537	0.08
JPEG	458698	459082	0.08
G711 encode	278578	278786	0.07
G711 decode	278331	278635	0.1

It is also interesting to study the impact of the proposed compilation techniques on the performance-oriented compiler optimization techniques. Figure 8 shows the percentage of hotspots of JPEG when compiled with the various optimization flags (-O0 to -O3) using the default and the proposed compilation flow. From this figure, it is clear that the number of hotspots is increased along with increasing the optimization level. However, the application of the proposed compilation flow is able to reduce the percentage of hotspots of the -O3 optimization level below the values found for -O0 using the default compiler. Therefore, the optimum selection for speed, size, and temperature is to use the -O3 optimization level along with the proposed compilation techniques. since it with this optimization flag, the code size is minimized and the performance is maximized. And after deploying the proposed compilation flow, the temperature is reduced. Therefore, all performance metrics obtain their best values using -O3 with the proposed compilation flow.

5. CONCLUSION

In this paper we have presented an efficient register-assignment mechanism that, based on a uniform distribution of accesses is able to optimize the thermal profile of the register file. The proposed technique, that has been embedded in a high-quality
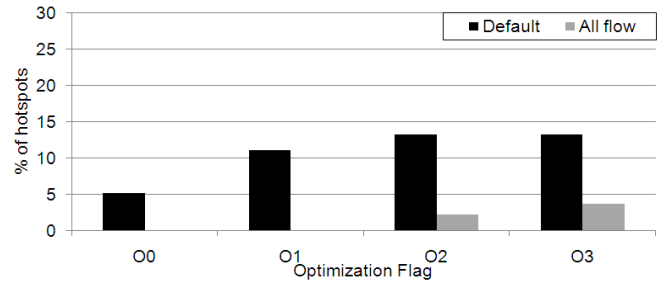

Figure 8: Percentage of hotspots effect with different optimization levels.

commercial compiler, is able to reduce the percentage the hotspots by 91% and the peak and mean temperature of the device up to 11%, without any impact on performance.

6. ACKNOWLEDGMENT

7. REFERENCES

[1] F. Fallah et al. Standby and Active Leakage Current Control and Minimization of CMOS VLSI Circuits. *IEICE Transactions on Electronics*, E88-C(4):509–519, 2005.

[2] O. Semenov et al. Impact of Self-Heating Effect on Long-Term Reliability and Performance Degradation in CMOS Circuits. *IEEE Transactions on Device and Materials Reliability (T-DMR)*, 6(1):17–27, March 2006.

[3] Zhijian Lu et al. Interconnect Lifetime Prediction for Reliability-Aware Systems. *IEEE Transactons on Very Large Scale Integration Systems (TVLSI)*, 15(2):159–172, 2007.

[4] National Semiconductor. Understanding Integrated Circuit Package Power Capabilities. www.national.com, April 2000.

[5] S. Borkar. Design Challenges of Technology Scaling. *Proceedings of IEEE Micro*, 19(4):23–29, 1999.

[6] Y. Li et al. Performance, Energy and Thermal Considerations for SMT and CMP Architectures. In *Proceedings of HPCA-11*, pages 71–82, 2005.

[7] Hushrav D Mogal et al. Thermal-Aware Floorplanning for Task Migration Enabled Active Sub-Threshold Leakage Reduction. In *Proceedings of ICCAD '08*, pages 302–305, 2008.

[8] Jayanth Srinivasan et al. Predictive Dynamic Thermal Management for Multimedia Applications. In *Proceedings of ICS '03*, pages 109–120, 2003.

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

[10] H. S. Kim et al. Adapting Instruction Level Parallelism for Optimizing Leakage in VLIW Architectures. *SIGPLAN Not.*, 38(7):275–283, 2003.

[11] Han-Saem Yun et al. Power-Aware Modulo Scheduling for High-Performance VLIW Processors. In *Proceedings of ISLPED '01*, pages 40–45, 2001.

[12] W. Zhang et al. Exploiting VLIW Schedule Slacks for Dynamic and Leakage Energy Reduction. In *MICRO 34*, pages 102–113, 2001.

[13] Jose L. Ayala et al. Energy-Aware Compilation and Hardware Design for VLIW Embedded Systems. *Inderscience International Journal of Embedded Systems*, 3(1):73–82, 2007.

[14] Madhu Mutyam et al. Compiler-Directed Thermal Management for VLIW Functional Units. *SIGPLAN Not.*, 41(7):163–172, 2006.

[15] Rajarshi Mukherjee et al. Temperature-Aware Resource Allocation and Binding in High-Level Synthesis. In *Proceedings of DAC '05*, pages 196–201, 2005.

[16] Sri Hari Krishna Narayanan et al. Compiler-Directed Power Density Reduction in NoC-Based Multi-Core Designs. In *Proceedings of ISQED '06*, pages 570–575, 2006.

[17] James Donald et al. Techniques for Multicore Thermal Management: Classification and New Exploration. In *Proceedings of ISCA '06*, pages 78–88, 2006.

[18] Kimish Patel et al. Active Bank Switching for Temperature Control of The Register File in a Microprocessor. In *Proceedings of GLSVLSI '07*, pages 231–234, 2007.

[19] Xiangrong Zhou et al. Temperature-Aware Register Reallocation for Register File Power-Density Minimization. *TODAES*, 14(2):1–22, 2009.

[20] Benjamin Carrion Schafer et al. Temperature-Aware Compilation for VLIW Processors. In *Proceedings of RTCSA '07*, pages 426–431, 2007.

[21] Anya Apavatjrut et al. Thermal Analysis of the Shared Register File in VLIW Architectures. In *Proceedings of PCI'07*, pages 165–175, 2007.

[22] *The SPARC Architecture Manual Version 8*. SPARC International, Inc, Menlo Park, CA.

[23] D. Atienza et al. HW-SW Emulation Framework for Temperature-Aware Design in MPSoCs. *TODAES*, 12(3):1–26, 2007.

[24] MediaBench Benchmark Suite. http://euler.slu.edu/ fritts/mediabench/.