Thermal-Aware Compilation for Register Window-Based Embedded Processors

Mohamed M. Sabry, José L. Ayala, and David Atienza

***Abstract*—The development of compiler-based mechanisms to optimize the thermal profile of large register files to improve the processor reliability 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. Register window-based architectures provide a relatively large register files. However, such large register files are not designed or utilized for thermal balancing or reliability enhancement. In this letter, we propose a compilation flow that utilizes the register windows to optimize the thermal profile and to reduce the hotspots. As a result, the thermal profile and reliability of the device is clearly improved. Simulation results show that the proposed flow achieves up to 91% reduction of hotspots and 11% reduction of the peak temperature in embedded processors.**

***Index Terms*—Compilers, registers, reliability, thermal management.**

I. 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.

- Leakage current, which presents an exponential dependence with temperature [1].
- Reliability of the system, since several processes are driven by the increase of temperature or the spatial and temporal gradients that appear during normal functioning.
- Timing delay variations, transient reduction in overall system performance, or even permanent damage in the devices [2] resulted from thermal evolution over a threshold in localized areas of the chip (hotspots).

It has been shown how the mean time between failure (MTBF) of an IC is divided by 10 for every 300° C rise in the junction temperature [3]. These facts explain the strong efforts being done nowadays in the area of thermal optimization in electronic circuits. Some of these efforts are being conducted to tackle the thermal problem at all levels of abstraction. Computer architects develop thermal efficient processor architectures that optimize the thermal behavior by proposing smart ways of sharing the computer resources [4]. Also, thermal-aware floorplanning is an intense area of research [5], since temperature depends on the placement of the units in the chip. 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 [6], and the code generation phase [7]. In this area, compilers play an important (and yet unexplored) role.

Due to its high utilization and relatively small area, the register file (RF) has been shown to have the highest peak temperature in several studies [8]. Reducing the percentage of the RF high-power-density spots leads to peak temperatures reduction for both the entire chip and the RF, which in turn results in improved reliability and reduced leakage power [9].

The thermal response of the RF 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. As we show in this letter, both parameters can be controlled by the compiler from a software perspective and this leads 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.

Overall, this letter proposes a novel thermal-aware compilation flow that is embedded in a state-of-the-art compiler. The proposed flow introduces a thermal-aware compiler that reallocates the registers based on application-specific information regarding 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 letter are the following:

- development of compilation techniques that improve the thermal profile of register window-based RF;
- integration of the proposed mechanisms in the CoSy compilation flow [10], a professional retargetable compiler;
- expanding the set of benchmarks conducted in [11] to cover a broader set of application types to assure the proposed flow robustness and effectiveness.

Simulation results show significant enhancements in terms of reduction of hotspots up to 91%, as shown in Section IV-A.

II. Related Work

In recent years, there has been an intense work at the compiler level in power-aware scheduling for VLIW processors, which

Manuscript received June 18, 2010; accepted August 16, 2010. Date of publication September 27, 2010; date of current version December 17, 2010. This work was supported in part by the EC-FP7 STREP Project (248776-PRO3D STREP), and the Spanish Government Research Grants TIN2008-00508 and CSD00C-07-20811. This manuscript was recommended for publication by R. Kumar.

M. M. Sabry and D. Atienza are with the Embedded Systems Laboratory (ESL), Ecole Polytechnique Fédérale de Lausanne (EPFL), 1015 Lausanne, Switzerland (e-mail: mohamed.sabry@epfl.ch; david.atienza@epfl.ch).

J. L. Ayala is with the Embedded Systems Laboratory (ESL), Ecole Polytechnique Fédérale de Lausanne (EPFL), 1015 Lausanne, Switzerland, and also with DACYA, Complutense University of Madrid, 28040 Madrid, Spain (e-mail: jayala@fdi.ucm.es).

Color versions of one or more of the figures in this letter are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/LES.2010.2081343

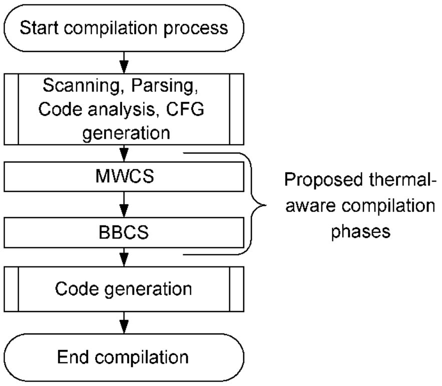
Start compilation process
Scanning, Parsing, Code analysis, CFG generation
MWCS
BBCS
Proposed thermal-aware compilation phases
Code generation
End compilation

Fig. 1. Proposed thermal-aware compilation flow.

proposed different approaches to turn off unused units to save leakage power [12]–[14]. Some of these works [15] explicitly target the RF for energy saving. However, these approaches do not consider temperature as the metric to be improved. Some of the first static approaches to thermal optimization are found in [6], [16], where load balancing heuristics and high-level synthesis techniques are considered. In Patel *et al.* [17], several thermal managing techniques for multicore architectures are explored, where register temperature reduction through both RF and register duplication are investigated.

Similar in scope to ours, a very recent work by Zhou *et al.* [18] proposes a register reallocation algorithm for power-density and thermal minimization in the RF. However, they target a few specific cases (high-power density registers) in VLIW architectures. VLIW architectures have been also considered in [19], where a thermal-aware instruction generation algorithm is proposed. However, VLIW architectures are not a window-based embedded processor, which is the architecture we focus on for typical embedded systems.

Our proposed work differs from previous static approaches in: 1) the minimization of several thermal- and reliability-related metrics like the mean and peak temperature of the RF, as well as the percentage of hotspots, with a negligible penalty; 2)the development of thermal-aware register reallocation techniques that cope with the limitations exhibited by RFs with register windows; and 3) the integration of these techniques in a high-quality industrial compilation flow.

We proposed a similar compilation flow in [11], which was proposed to a broad set of processing architectures. However, this letter differs from our previous work in: 1) a more focused analysis to register window-based embedded processors; and 2) the evaluation of the proposed flow effectiveness using an expanded set of benchmarks.

III. Thermal-Aware Register Assignment Flow

Registers in RFs exhibit a high thermal profile due to high frequency of accesses and proximity of hot registers. Based on the thermal traces collected for different benchmarks, it is observed that the thermal profile of the device is improved when the registers are assigned from spread spots of the RF, reducing in this way the mutual diffusion effect [20]. Based on this observation, register reallocation policies should be designed in order to allocate physically non-adjacent registers.

Thus, a new thermal-aware compilation flow (Fig. 1) is proposed to minimize the high thermal profile of the RF. Such flow is designed to minimize the mutual diffusion, as well as self-heating effects. The proposed flow targets register window-based or register bank-based processing architectures.

The proposed flow is integrated within the compilation process, before code generation or code emission process, where the pseudocode generated in the previous phases is translated to target code. This flow is divided into two stages: *multiwindow context switching (MWCS)* and *basic-block code splitter (BBCS)*, which we describe next.

A. Multiwindow Context Switching (MWCS)

The MWCS technique aims to reduce the mutual thermal diffusion between two adjacent windows allocated due to functional (or subfunctional) calls. Normally, the hierarchy of any program includes a group of functions, and such hierarchy has a certain depth that indicates the number of nested called functions (subfunctions). For a function F that is in the hierarchy and is allocated to register window i, the subfunction in the proceeding level is executed in window $i-1$. This leads to the usage of two adjacent windows that have a thermal diffusion impact on each other. Hence, the overall thermal profile of the RF is impacted negatively.

MWCS is proposed such that successive levels in a program functional hierarchy are executed in nonadjacent windows, hence the thermal diffusion between register windows is diminished in such a scenario. The proposed technique shifts from the working register window (i) to a new one (k), in case of a functional call.

For an RF having N register windows, we calculate k using

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

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

This technique ameliorates the sequence in which register windows are utilized, such that the spatial distance between two consecutively used windows is increased, as well as the temporal separation between two physically adjacent windows.

Applying this technique has an overhead cost, since additional instructions are needed for such window movements. However, this overhead is negligible, as we show in Section IV-A.

B. Basic-Block Code Splitter (BBCS)

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 subfunctional calls in such function (subfunctions uses another window other than the allocated ones). This technique allows 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 is executed using register window i, and the rest is executed using $i-1$). In addition to thermal reduction benefits, BBCS aims to enhance the bias temperature instability (BTI) by toggling more cells by using more register windows, which in turn enhances the system reliability [21].

The algorithmic flow of BBCS is briefly shown in Algorithm III.1. This technique explores the whole procedure via its control flow graph (CFG) to locate a point in the procedure structure, where the preceding instructions are dead (are not executed again) and the proceeding ones must be executed regardless the state of the variables (i.e., no flow control instructions are preventing such instructions from being executed).

Starting from the entry block, for each processed basic block B_P, the predecessor blocks are checked to be in $Predecessor\_list$. If any block B_L is not in this list, it implies that such block is a successor block and is included in an iterative loop with B_P. Hence, B_L is inserted in another list named $Notfound\_list$. When the predecessor blocks are processed, the successor blocks are inserted in $Successor\_list$ (if it is not already there) and checked to be in $Notfound\_list$. If any block is found, its index is removed from that list.

When the splitting condition is satisfied, the compiler counts the number of input live registers that should be available in the new window (N_{liveR}). If N_{liveR} is lower than a certain threshold (TH), then the splitting occurs. If not, the algorithm continues looking for another splitting point. This is shown in the algorithm by a call to MAY_SPLIT that returns a flag indicating if the above mentioned criteria is fulfilled or not (1 or 0). When splitting occurs, a microcode is injected to move the live registers to the new window, in addition to the context switching instruction. It is worth mentioning that the scope of live registers here is within the local variables, since global variables are promoted to the main memory, hence not affected by the change of the used window.

The mentioned threshold (TH) is related to the number of *output* registers of the RF, N_{OR}, and the remaining number instructions after the potential splitting block, N_{iB}. TH is calculated using (2).

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

This equation can be interpreted as follows: the instruction overhead resulting from moving the live registers from the old window should not exceed 10% the number of proceeding instructions; N_{iB}. The overhead resulting from moving one live register is two instructions; one instruction is required for moving the live register to an output register, while the other is executed after switching to move the input register (output of the previous window) to its proper location.

Algorithm III.1: BBCS($CFG\ code\_CFG$)

```
$Predecessor\_list = []$
$Notfound\_list = []$
$Successor\_list = [code\_CFG.entry\_block]$
$SPLIT = 0$
while $SIZE(Successor\_list > 0)$ and $SPLIT == 0$
do
    $B_P = FIRST\_BLOCK(Successor\_list)$
    $DELETE\_FIRST\_BLOCK(Successor\_list)$
    $L_1 = GET\_PREDECESSOR(B_P, code\_CFG)$
    for each $B_L \in L_1$
    do
        if $B_L \notin Predecessor\_list$
            then $INSERT(B_L, Notfound\_list)$
    $INSERT(B_P, Predecessor\_list)$
    $L_2 = GET\_SUCCESSOR(B_P, code\_CFG)$
    for each $B_L \in L_2$
    do
        if $B_L \in Notfound\_list$
            then $DELETE(B_L, Notfound\_list)$
        if $B_L \notin Predecessor\_list$ and
        $B_L \notin Successor\_list$
            then $INSERT(B_L, Successor\_list)$
    if $SIZE(Successor\_list) == 1$ and
    $IS\_EMPTY(Notfound\_list)$
    then
        $B_S = FIRST\_BLOCK(Successor\_list)$
        $SPLIT = MAY\_SPLIT(B_S, code\_CFG)$

if $SPLIT == 1$
    then return $(FIRST\_BLOCK(Successor\_list))$
    else return $(NULL)$
```

Moreover, the instruction overhead is limited by the available *output* registers in the window N_{OR}, because this is an architectural-based limitation and it is not efficient to use the memory to move the live registers between the used windows.

IV. Case Study: SPARC V8

The SPARC V8 architecture has been selected as representative case study of 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 RFs. Register windows are found only in the integer, while the floating point RF is a single window with 32-register RF. The considered SPARC processor contains an 8 window RF that includes 136 registers overall (8 *global* and 16 registers per window).

A. Simulation Results

The experimental work conducted has been performed using the hardware-software (HW-SW) emulation platform presented in [20]. This platform is required to extract the power traces corresponding to the execution of the application.

The proposed compilation techniques have been embedded in the professional CoSy compilation framework provided by ACE [10]. A set of benchmarks have been used to measure the proposed flow performance. Such benchmarks were selected to explore various application types [multimedia [23], signal processing (FFT and SPLINE], and conventional benchmarks (Dhrystone)). The proposed compilation flow was applied on the main application, unlike the system libraries, which is compiled using the default compilation flow. Such compilation procedure is conducted based on the fact that thermal-aware com-

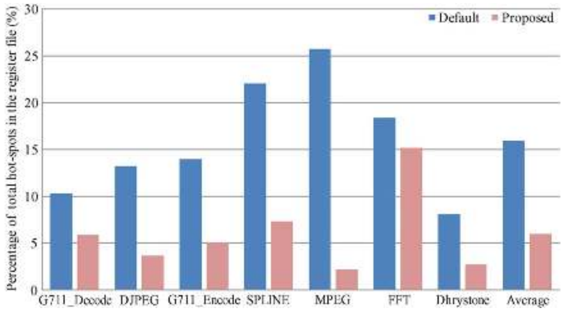

Fig. 2. Percentage of hotspots on different applications using the original and modified compilers.

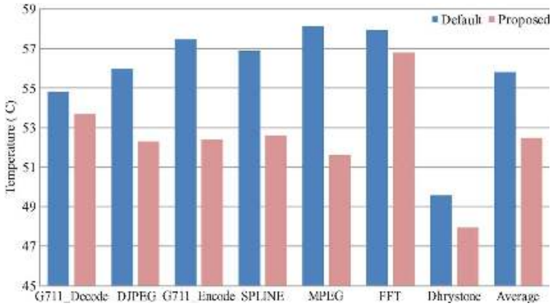

Fig. 3. Peak temperature values of different applications using the original and modified compilers.

piling the system libraries is application dependent, implying to rebuild them for every application.

Simulation results show that the proposed compilation flow achieves a significant reduction in both the percentage of hotspots and the peak temperature, as shown in Figs. 2 and 3, respectively. On average, the percentage of hotspots is reduced by 63% 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. However, the peak temperature of FFT is not much affected (reduced by 1.5%). This is related to the structure of this program. FFT has a hierarchy depth of 3 levels and the dominant cause of thermal evolution is the computation performed on a single function named fft_float. This function is a single loop function that could not be split using BBCS. Hence, the only benefit obtained here is the minimization of the diffusion effect between windows.

V. Conclusion

In this letter, we have presented a thermal-aware compilation flow that, based on a uniform distribution of accesses is able to optimize the thermal profile of the register file, hence enhance the reliability of the processor. This flow has been embedded in a high-quality commercial compiler, and is able to reduce the percentage of hotspots as well as the peak temperature of the device up to 91% and 11%, respectively, without any impact on execution time.

Acknowledgement

The authors would like to thank Associated Compiler Experts (ACE) for their licenses donation of the CoSy Compilation Framework.

References

[1] F. Fallah *et al.*, “Standby and active leakage current control and minimization of CMOS VLSI circuits,” *IEICE Trans. Electron.*, vol. E88-C, no. 4, pp. 509–519, 2005.

[2] O. Semenov *et al.*, “Impact of self-heating effect on long-term reliability and performance degradation in CMOS circuits,” *IEEE Trans. Device Mater. Rel.*, vol. 6, no. 1, pp. 17–27, Mar. 2006.

[3] National Semiconductor, Understanding Integrated Circuit Package Power Capabilities [Online]. Available: www.national.com Apr. 2000

[4] Y. Li *et al.*, “Performance, energy and thermal considerations for SMT and CMP architectures,” in *Proc. Int. Symp. High-Perform. Comput. Arch. (HPCA-11)*, San Francisco, CA, 2005, pp. 71–82.

[5] H. D. Mogal *et al.*, “Thermal-aware floorplanning for task migration enabled active sub-threshold leakage reduction,” in *Proc. Int. Conf. Comput.-Aided Design (ICCAD)*, San Jose, CA, Nov. 2008, pp. 302–305.

[6] R. Mukherjee *et al.*, “Temperature-aware resource allocation and binding in high-level synthesis,” in *Proc. Design Autom. Conf. (DAC)*, Shanghai, China, 2005, pp. 196–201.

[7] F. Mulas *et al.*, “Thermal balancing policy for multiprocessor stream computing platforms,” *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 28, no. 12, pp. 1870–1882, Dec. 2009.

[8] J. Srinivasan *et al.*, “Predictive dynamic thermal management for multimedia applications,” in *Proc. Int. Conf. Supercomput. (ICS)*, San Francisco, CA, 2003, pp. 109–120.

[9] J. L. Ayala *et al.*, “Thermal-aware data flow analysis,” in *Proc. Design Autom. Conf. (DAC)*, San Francisco, CA, 2009.

[10] ACE Cosy Compiler [Online]. Available: http://www.ace.nl/compiler/cosy.html

[11] M. M. Sabry *et al.*, “Thermal-aware compilation for system-on-chip processing architectures,” in *Proc. Great Lakes Symp. VLSI*, Providence, RI, 2010, pp. 221–226.

[12] H. S. Kim *et al.*, “Adapting instruction level parallelism for optimizing leakage in VLIW architectures,” *SIGPLAN Not.*, vol. 38, no. 7, pp. 275–283, 2003.

[13] H.-S. Yun *et al.*, “Power-aware modulo scheduling for high-performance VLIW processors,” in *Proc. Int. Symp. Low Power Electron. Design (ISLPED)*, Huntington Beach, CA, 2001, pp. 40–45.

[14] W. Zhang *et al.*, “Exploiting VLIW schedule slacks for dynamic and leakage energy reduction,” in *Proc. 34th Annu. Int. Symp. Microarch. (MICRO 34)*, Austin, TX, 2001, pp. 102–113.

[15] J. L. Ayala *et al.*, “Energy-aware compilation and hardware design for VLIW embedded systems,” *Indersci. Int. J. Embed. Syst.*, vol. 3, no. 1, pp. 73–82, 2007.

[16] M. Mutyam *et al.*, “Compiler-directed thermal management for VLIW functional units,” *SIGPLAN Not.*, vol. 41, no. 7, pp. 163–172, 2006.

[17] K. Patel *et al.*, “Active bank switching for temperature control of the register file in a microprocessor,” in *Proc. Great Lakes Symp. VLSI*, Lausanne, Switzerland, 2007, pp. 231–234.

[18] X. Zhou *et al.*, “Temperature-aware register reallocation for register file power-density minimization,” *ACM Trans. Design Autom. Electron. Syst.*, vol. 14, no. 2, pp. 1–22, 2009.

[19] B. C. Schafer *et al.*, “Temperature-aware compilation for VLIW processors,” in *Proc. 13th IEEE Int. Conf. Embed. Real-Time Comput. Syst. Appl.*, Daegu, Korea, 2007, pp. 426–431.

[20] D. Atienza *et al.*, “Reliability-aware design for nanometer-scale devices,” in *Proc. Asia South Pacific Design Autom. Conf. (ASP-DAC)*, Seoul, Korea, 2008.

[21] S. V. Kumar *et al.*, “Impact of NBTI on SRAM read stability and design for reliability,” in *Proc. 13th IEEE Int. Conf. Embed. Real-Time Comput. Syst. Appl.*, Daegu, Korea, 2006.

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

[23] MediaBench Benchmark Suite [Online]. Available: http://euler.slu.edu/fritts/mediabench/