Analysis of the Thermal Impact of Source-Code Transformations in Embedded-Processors

José L. Ayala, Cándido Méndez, Marisa López-Vallejo
Departamento de Ingeniería Electrónica
Universidad Politécnica de Madrid (Spain)
Email: {jayala, cmendez, marisa}@die.upm.es

***Abstract*— This paper makes use of a *black box* modeling approach to analyze the thermal behavior of microprocessor systems. Our goal is to study the impact of code transformations in the thermal behavior of the chip. The analysis of the thermal effect of the source code modifications can be included in a temperature-aware compiler which minimizes the total temperature of the chip, as well as the temperature gradients, according to these guidelines.**

I. Introduction

As technology scales, higher power consumption coupled with smaller chip area results in higher power density, which in turn leads to higher temperature in the chip [1]. In fact, extrapolating the changes in microprocessor organization and the device miniaturization, future power density can be forecasted to $200W/cm^2$ [2]. This requires extensive efforts on cooling techniques which have shown to be complex and highly expensive.

While hardware solutions to temperature management problems are very important, software can also play an important role because it determines which circuit components are used during the execution and for how long. In particular, compilers and source code transformations determine the data and instruction access patterns of applications, what can shape the power density profile.

In this work, we analyze the effect of different code-level transformations in the thermal map. The use of a thermal model and analysis tool allows us to characterize the thermal behavior of the functional units in the microcontroller when different implementations of the same benchmark are executed. Moreover, the analyzed code transformations can be included in a thermal-aware compiler whose objective is to reduce the gradient of temperature in the chip area of the target architecture.

Nowadays there has been an increasing interest to provide a detailed die temperature distribution [3], [4]. In these works, the authors present different detailed full chip thermal models. However, it has been reported recently that the results achieved by these models may present inaccuracy problems [5], [6].

In the field of temperature optimization, research has mainly focused on runtime techniques. These approaches use predictive algorithms [7], dynamic profiling [1] or activity migration [8] to reduce the temperature in a chip by dynamically distributing jobs on multiple processors. Our work is different from these in that it is static in its approach and targets the effect of source-level modifications in the thermal behavior. The related work in the area of static thermal management is still scarce and few works can be found targeting the temperature-aware floorplanning of the chip [9] and the loop parallelization for multiprocessors [10]. However, this is the first work which analyzes the effect of code transformations in the thermal behavior of the functional units of the chip, providing an expertise that can be incorporated in a temperature-aware compiler.

In summary, the contributions of this paper are:

1. Analysis of the effect of several source-code transformations in the thermal behavior of the chip. The analysis is not only focused on the total temperature of the chip but it also takes into account the gradients of temperature that can appear between the functional units.

2. Settlement of the basis for a temperature-aware compiler.

This paper is composed as follows: Our proposed methodology is briefly explained in Section II. Section III covers the conducted experimental works and finally, some conclusions are drawn in Section IV.

II. Methodology

In this work we use our thermal parameterization methodology[1] [11] to evaluate the impact of the source code modifications in the temperature distribution of a low-cost microcontroller when running a specific application program.

The reason for selecting a low-cost microcontroller as our target processor is that nowadays this kind of microcontrollers are widely used in many low temperature applications. Moreover, their reduced instruction set and their simple architecture, with few easy-to-distinguish functional units, facilitates the task of characterizing the whole architecture blocks.

One of the issues when performing a thermal characterization of these microcontrollers is the lack of information about their layout and the implementation of their functional units. Another problem is the difficulty of measuring the temperature of the chip, what requires expensive external equipment. We have avoided these problems devising a *black box* characterization which does not require this information, and using power as an intermediate parameter. The power consumption is a variable that can be easily measured in the

[1]The explanation of the whole thermal characterization methodology is out of the scope of this paper. Here a brief summary is presented to settle the basis of the thermal analysis carried out in this work.

processor without expensive and complex equipment. In the following we briefly describe the devised methodology for the thermal characterization.

1) **Power Characterization:** The first step on our methodology is to obtain the power consumed by all functional units of the microcontroller by means of electrical measures. To collect these power figures we have developed a set of benchmarks that are able to excite the processor in a particular way.
   Once we have gathered power measures from all the benchmarks, it is possible to obtain a set of power/utilization equations to estimate the power behavior of the microcontroller for different applications.
2) **Thermal parameterization:** With all the power figures collected, the next step is to obtain the temperature estimation. We have used a dynamic thermal model based on RC-networks. With this thermal model, each functional block is modelled as an RC pair, representing the vertical heat transfer.
   As a result of this process, a relationship between temperature and utilization is obtained for each functional unit. After this, the thermal evolution for every functional unit, as well as for the whole chip, can be studied.
3) **Application of the model:** Finally, the characterization presented in the previous sections has been used to analyze the thermal effect of the source modifications in an image filtering benchmark. We have developed various implementations of this benchmark with several code-level transformations. Doing this, it can be observed how the different code variations affect the temperature distribution inside the chip. Examining the results, the designer can reach conclusions such as which transformations should be selected to reduce the total temperature in the chip, or which transformations achieve the best temperature distribution minimizing the gradients. These conclusions could be then included in a temperature-aware compiler.

Summarizing, in precedent sections we have presented the methodology followed to accurately characterize the thermal behavior of a low-cost microcontroller. With all the parameters gathered, it is possible to simulate the thermal variances of the chip and its functional blocks. Besides, this simulation could be easily fully automated. Moreover, the methodology is target-independent. A disadvantage of this characterization method is the difficulty of reaching a lower level of granularity. This is imposed by the simplicity of the microcontroller and the lack of knowledge about the functional units implementation. A selection of a processor with a richer architecture would allow to increase the granularity of this analysis.

III. Experimental Work

For our set of experiments we have selected the Microchip PIC16F873, which satisfies the desired complexity and can be found in many embedded systems with low-power and low-temperature constraints. This device is a low-cost microcontroller with a Harvard RISC CPU. It includes 4K x 14 words of FLASH program memory, 192 bytes of RAM and a data EEPROM of 128 bytes. It also includes an SRAM Register file (RF), a Program Counter (PC) with an eight level deep hardware stack and an 8-bit ALU with an accumulator (ACC). As can be seen, the architecture resembles the common characteristics of a broad range of embedded processors.

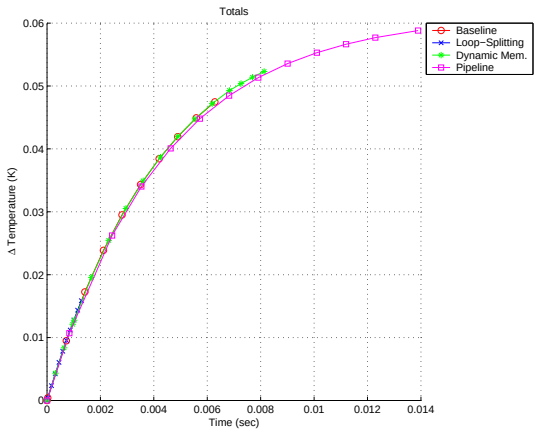

Fig. 1. Evolution in time of the total temperature in the chip

In order to analyze the thermal behavior of the PIC when applying a source-code transformation, an image processing algorithm has been chosen[2]. This benchmark stresses the processor architecture in a similar way to common embedded applications. Due to the severe memory constraints imposed by the PIC architecture, this algorithm has been adapted to satisfy all the memory requirements.

We have developed four different versions of the image filtering benchmark according to different source-level transformations. These four implementations are: Baseline, Loop-splitted, Dynamic memory and Pipelined.

The thermal characterization methodology has been used to perform a global (total temperature) and local (temperature gradient) experimental work. The global and local thermal analysis of the previous implementations of the image filtering algorithm will provide a complete knowledge of the thermal impact of the source-level transformations.

A. Global Analysis

On a first place, the impact of the code modifications in the temperature of the whole chip has been analyzed. This analysis provides a general view on the temperature evolution of the chip depending on the benchmark implementation. This kind of global heating has a negative effect on propagation delays, signal integrity and power consumption.

Figure 1 shows the evolution in time of the total temperature in chip when the four different implementations are executed. It can be observed that there are few differences in the way that the temperature behaves in time for all the implementations.

[2]The benchmark is an image filtering application which is based on a loop dominated algorithm as many other multimedia applications.

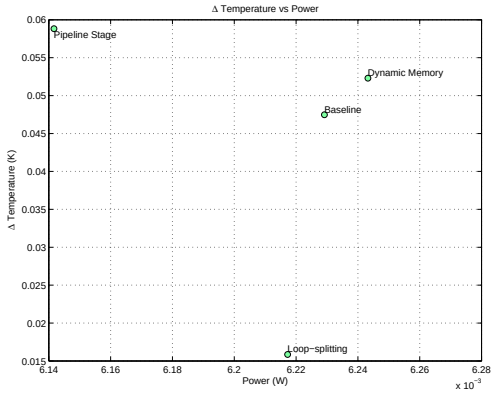

Fig. 2. Power consumption vs. Total temperature in the chip

We can conclude that the studied code transformations do not have a representative impact on the global behavior of the temperature in the chip. However, it is clear that there are absolute temperature differences between implementations that would increase in more complex architectures and applications. Such differences are caused by the execution time required to finish the applications.

For example, the loop-splitted code is executed seven times faster than the pipelined implementation. This speedup in the execution causes that the chip does not reach the highest temperatures, which appear during the execution of other implementations. Thus, the loop-splitted implementation can be considered a low-temperature transformation.

Figure 2 shows the power consumed and the temperature reached in the chip during the execution of the four different implementations. The loop-splitted implementation is able to reduce the total temperature in the chip by a 66% with respect to the baseline implementation without an impact on the power consumption. However, the pipelined implementation decreases the power consumed by the application but presents a significant increase in the temperature. Thus, to optimize for low-temperature does not mean to optimize for low-power.

B. Local Analysis

When executing an application, not all the functional units are used at the same time, and probably workloads are different between them. This leads to different temperatures on different points of the chip which is translated into temperature gradients. If only the global warming of the chip is observed, these temperature gradients can be probably overlooked and cause several problems to the chip. High temperature gradients can cause electromigration or thermal diffusion on functional units. Thus, analyzing what functional units are warmer during the execution of an application and the consequent temperature gradients, it is possible to establish techniques and rules that could be included into a temperature-aware compiler to make temperature inside the chip more uniform.

Figure 3 presents the thermal evolution of all the functional units considered in the microcontroller when executing the Dynamic Memory implementation of the benchmark. The values of temperature have been calculated with respect to the

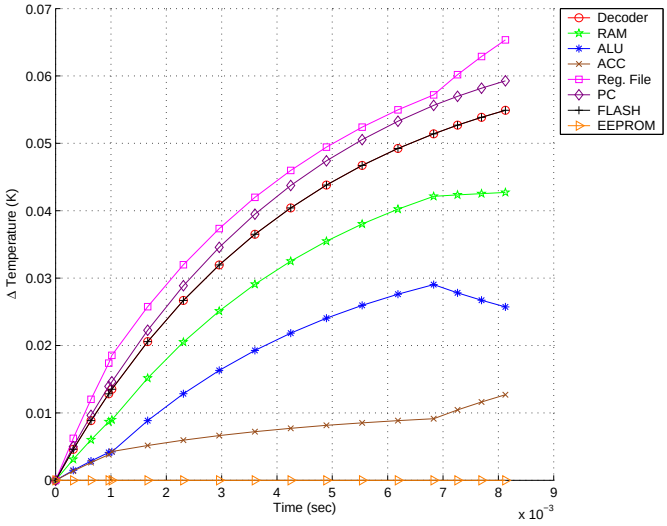

Fig. 3. Local thermal evolution for the Dynamic Memory implementation

ambient temperature (relative values in the figures). As can be seen, RF, PC, FLASH instruction memory and Decoder unit show the highest temperature values. This behavior can be explained, on the one hand, because FLASH, PC and RF are used every cycle when fetching a new instruction. On the other hand, the Decoder unit is always used before executing a new instruction. The zero value of the EEPROM unit shows that it is not used during the execution, so it maintains its initial temperature.

To study the effects of the code transformations on the thermal evolution of different functional units and their impact on the chip, we have considered two types of thermal gradients: First, we have studied the maximum gradient that can appear in the chip, that is, the worst case gradient. Then, we have analyzed a typical gradient.

1) Worst Case Analysis: The maximum expected temperature gradient in the chip is the temperature difference between its hottest and its coolest units. Observing this gradient, it is possible to find the worst thermal stress into the chip and apply the source code modifications in a way that this effect is softened. Figure 4 shows maximum gradients for the four implementations of the benchmark. It can be seen that in the pipelined implementation there is an important temperature gradient between the RF and the Working register (ACC). In the rest of the implementations, their maximum gradients are much lower. The reason for these differences is that transfers between different memory spaces are massive in the pipelined implementation. These transfers are made using indirect addressing, what makes use of some special registers located in the RF. This massive use of registers causes a higher heat-up of the RF block.

Thus, the pipelined implementation can be applied to a reduced portion of the source code (local optimization) and the effect of the temperature gradient is minimized as being restricted to a shorter execution time. Also, an alternative source code transformation that cools down these units can be employed in parallel with the pipelined code modification.

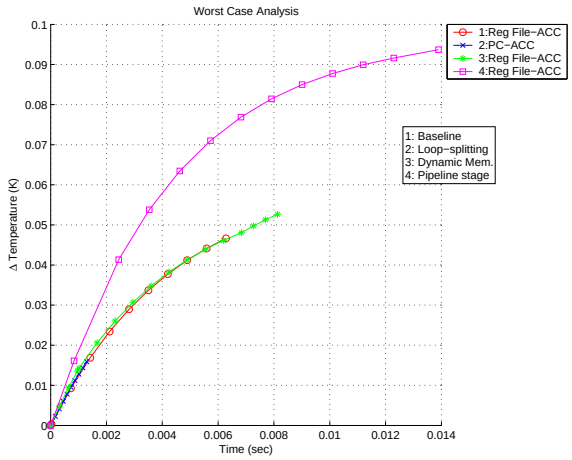

Fig. 4. Maximum expected temperature gradients

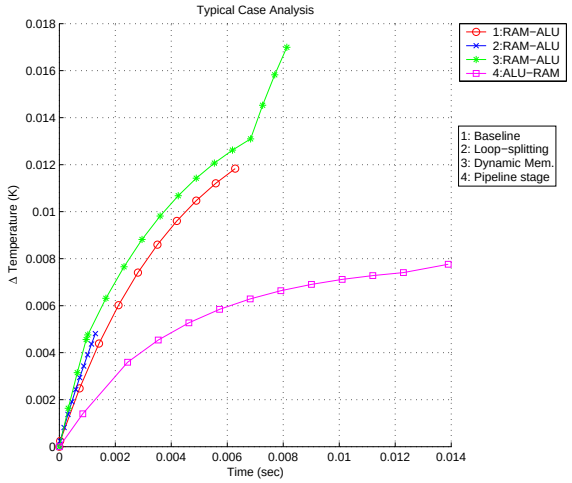

Fig. 5. Typical expected temperature gradients

2) Typical Case Analysis: It is also interesting to analyze the thermal gradient existing between functional units that do not have extreme temperatures. This will be the situation of most of the blocks during the execution of a benchmark and needs to be characterized properly. Figure 5 shows the typical expected temperature gradients for the four implementations of the benchmark. In this case, the dynamic memory implementation presents the highest temperature gradient due to the amount of memory transactions, highly increased with this transformation.

This figure presents other interesting results. The gradient profile of the dynamic memory implementation shows one quick gradient variation. This variation coincide with the operation of memory release. During these operations, the ALU has little use in favor of the memory transfers. This analysis can be taken into account in order to propose a scheduling of source code transformations that homogenize the thermal behavior in the functional units.

An example of this homogenization policy is shown in Figure 6. As can be seen, the combination of the loop-splitting transformation with the use of dynamic memory provides a

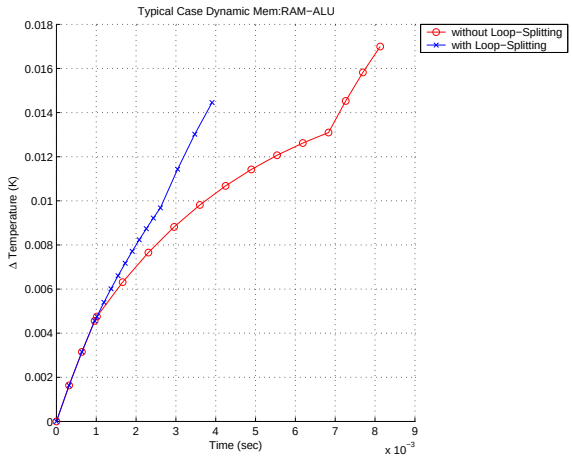

Fig. 6. Typical expected temperature gradients before and after homogenization policy

better response in terms of thermal gradients. The use of the loop splitting transformation produces a heating in the RAM of the system at the beginning, which reduces the transient in the temperature when the memory begins to be stressed.

IV. Conclusions

This paper has presented an efficient characterization of the thermal effects incurred by different source code transformations. The characterization methodology is target-independent and does not require a deep knowledge of the implementation. The analysis of these effects is a precious information to be considered for achieving the temperature-aware compilation.

Our ongoing work is currently focused on increasing the set of source code transformations and looking for an analytical way to apply this transformations in the thermal homogenization problem.

References

[1] D. Brooks and M. Martonosi, "Dynamic thermal management for high-performance microprocessors," in *HPCA*, 2001.

[2] http://www.hpl.hp.com/research/dca/smart_cooling/.

[3] H. Su, F. Liu, A. Devgan, E. Acar, and S. Nassif, "Full leakage estimation considering power supply and temperature variations," in *ISLPED*, 2003.

[4] P. Li, L. Pileggi, M. Ashegi, and R. Chandra, "Efficient full-chip thermal modeling and analysis," in *ICCAD*, 2004.

[5] W. Huang, E. Humenay, K. Skadron, and M. Stan, "The need for a full-chip and package thermal model for thermally optimized IC designs," in *International Symposium on Low Power Electronics and Design*, 2005.

[6] W. Huang, M. Stan, and K. Skadron, "Parameterized physical compact thermal modeling," *IEEE Trans. on Component Packaging and Manufacturing Technology*, vol. 28, no. 4, pp. 615–622, December 2005.

[7] J. Srinivasan and S. V. Adve, "Predictive dynamic thermal management for multimedia applications," in *ICS*, 2003.

[8] S. Heo, K. Barr, and K. Asanovic, "Reducing power density through activity migration," in *ISLPED*, 2003.

[9] K. Sankaranarayanan, S. Velusamy, M. Stan, and K. Skadron, "A case for thermal-aware floorplanning at the microarchitectural level," *Journal of Instruction Level Parallelism*, vol. 7, October 2005.

[10] S. H. K. Narayanan, G. Chen, M. Kandemir, and Y. Xie, "Temperature-sensitive loop parallelization for chip multiprocessors," in *ICCD*, 2005.

[11] C. Méndez, J. L. Ayala, and M. López-Vallejo, "Target independent thermal modeling for embedded processors," in *IES*, 2006.