

Instruction and Data Memory Energy Trade Off

Tom Vander Aa, Geert Deconick, Henk Corporaal
KULeuven/ESAT, Kasteelpark Arenberg 10, 3001 Leuven
<Tom.VanderAa@esat.kuleuven.ac.be>

September 1, 2004

Abstract

This report proposes a methodology to explore the transfer and storage of both instructions and data in the memory hierarchy of low energy embedded processors. By carefully trading off the energy in both memory types we are able to minimize the total system energy. For this report we have optimized for one application, a medical imaging application, both the instruction and data memory side. Results show a reduction in the total memory energy of 80%.

1 Introduction

Low energy is a key component of embedded processors these days. It is shown that when running embedded applications on such processors a significant amount of energy is spent in the memories, both the instruction memories (storing the program being run) and the data memories (storing the data being manipulated by the program). For both types of memory, methods exist that optimize the energy consumption. (See for example [2, 5] for the data memories or [3, 1] for the instruction memories.) These methods, however, have been developed and deployed mostly independently, and their interference has seldom been studied.

For this report we have optimized for one application, a medical imaging application, both the instruction and data memory side. For the data memory side we used the DTSE [2] methodology. For the instruction side, a similar path was followed. The most important steps are:

1. Platform Independent Part

- (a) Apply loop transformations to optimize the data memory energy.
- (b) Apply loop transformations to optimize the instruction memory energy, but with-

out changing the behaviour of the data memory.

Results that will be shown for this part, are estimated lower bounds.

2. Platform Dependent Part

- (a) Choose the optimal data and instruction memory hierarchy for this application and decide what parts of the data/instructions should be mapped to what layer in the hierarchy.
- (b) Implement the chosen mapping on a real processor as efficiently as possible.

In the next two sections the result of these two steps for the cavity detection application will be presented.

2 Platform Independent Loop Transformations

2.1 Initial Implementation of the Cavity Detection Algorithm

Originally the Cavity Detection Algorithm was implemented as a chain of independent filters (Figure 1a). The first filter took the input image and did a filter operation on the complete image (in this case a vertical/horizontal Gaussian blur), producing a new output image. Each of the next filters operated on the output of the previous filter. Since each time a new image was produced, and since neither the new nor the original image fitted in the internal memory of the processor, all the data memory accesses related to the image went to the external memory. This had a huge energy and time penalty, as can be seen in Figure 2 (initial). Since the code base of each of the filters was small, the instruction memory energy was low compared to the data memory energy.

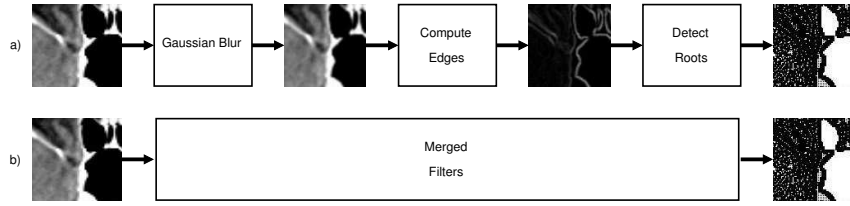


Figure 1: Structure of the Cavity Detection Algorithm; a) implementation before DTSE; b) implementation after DTSE

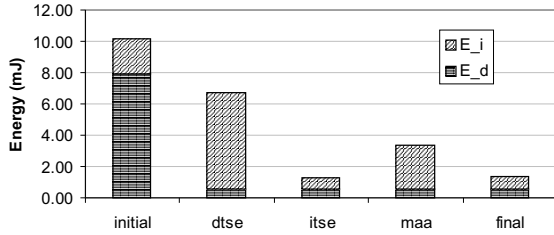


Figure 2: Instruction and Data Memory Energy for Optimal Hierarchies; estimates for the versions before DTSE (initial), after DTSE (dtse), after ITSE (itse); measured values for the unoptimized (maa) and optimized (final) mapping on an existing platform

2.2 Data Memory Code Transformations

DTSE was applied and the filter chain was merged (Figure 1b) such that each of the filters operated on only a little part of the image, just enough for the next filter to start, this way eliminating effectively all the data accesses to the background memory. Energy reduction on the data side was dramatic (Figure 2, dtse). On the instruction side, however, this became worse. Because of the merging, different small loops with good locality (small working set) became one big loop. The optimal size of instruction level 1 memory grew from 64 to 256 bytes. At the same time extra instructions were added: checks in the code were needed to make sure that the data dependencies were still correct, i.e. to make sure the next filter in the pipeline was not started too early. The net result was only a slight decrease in total energy.

2.3 Instruction Memory Code Transformations

After DTSE, several source code transformations were applied to the cavity detector implementa-

tion to improve instruction memory energy (without worsening data memory energy). Three ways can be investigated to reduce energy:

1. Reduce the number of operations. This will not only reduce the number of access to the instruction memory but will also reduce the runtime of the application, which is, in general, very beneficial for energy.
2. Increase the parallelism. This will both increase performance and reduce the number of useless NOP-operations.
3. Increase the locality of the instructions, i.e. reduce the size of the loop bodies.

Transformations that were applied are: function inlining, loop unrolling and software pipelining, hyperblock creation and loop body split. Figure 2 (itse) shows how these transformations have reduced the instruction memory energy.

3 Platform Dependent Optimizations

Since all previous steps were platform independent, the results that were presented were only lower bound estimates. In this section we will choose a real platform and map the application in this platform.

3.1 Platform Exploration and Assignment

To find the best platform, we allow the memory sizes of our processor to vary according to the template shown in Figure 3. Next, we find what is the best instance of this template and the best mapping of the application on the instance.

To implement the latter, we need to copy the right parts of the data and instructions in the right memory

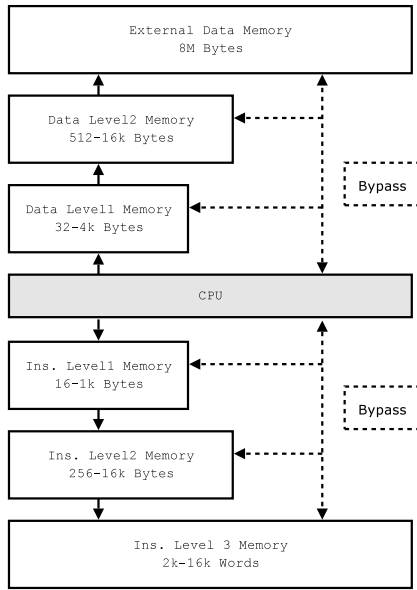


Figure 3: Memory Architecture Design Template

at the right time. If we do this badly, the net result will be an energy increase instead of decrease (with respect to the earlier estimates), due to the large overhead of making the copies (Figure ??, maa) and the added complexity in the address calculations [4].

3.2 Optimized Implementation

By using hardware helpers, such as DMA controllers and by mapping parts of the data to the register file, we can already reduce this part of the overhead

The other type of overhead is caused by modulo operations in the address calculations. By rounding the buffer sizes to the nearest power of two, the modulo operation is replaced with a logical and operation. In our case the line buffers of the three filters was rounded up from 1280 bytes to 2K bytes. Clearly this imposes a trade-off between the complexity of the indexing and the memory usage. The impact of this decision depends on the fact that the buffers still fit in the internal memory of the processor, or not.

4 Final Result

The final result is an implementation of the cavity detection algorithm that is both optimized for data and instruction memory energy and can be efficiently implemented on an programmable platform.

For more detailed information see [6].

References

- [1] Nikolaos Bellas, Ibrahim Hajj, Constantine Polychronopoulos, and George Stamoulis. Architectural and compiler support for energy reduction in the memory hierarchy of high performance microprocessors. In *Proc of ISLPED*, August 1998.
- [2] Francky Catthoor, Koen Danckaert, Chidamber Kulkarni, Erik Brockmeyer, Per Gunnar Kjeldsberg, Tanja Van Achteren, and Thierry Omnes. *Data access and storage management for embedded programmable processors*. Kluwer Academic Publishers, March 2002.
- [3] Lea Hwang Lee, William Moyer, and John Arends. Instruction fetch energy reduction using loop caches for embedded applications with small tight loops. In *Proc of ISLPED*, August 1999.
- [4] Miguel Miranda, C. Ghez, Chidamber Kulkarni, Francky Catthoor, and Diederik Verkest. Systematic speed-power memory data-layout exploration for cache controlled embedded multimedia applications. In *Proc. ACM Design Automation and Test in Europe Conf.*, pages 9–13, march 2000.
- [5] Preeti Ranjan Panda, Nikil D. Dutt, and Alexandru Nicolau. *Memory Issues in Embedded Systems-On-Chip – Optimizations and Exploration*. Kluwer Academic Publishers, Boston, October 1998.
- [6] Tom Vander Aa. Energy optimizations of data and instruction memories on a medical imaging application. Technical report, ESAT/ELECTA, K.U.Leuven, Belgium, September 2004.