Automatic Design and Exploitation of Memory Hierarchies for Efficient Embedded Systems

Kristof Beyls

Abstract— **In many computer systems, a large portion of the execution time and energy consumption is due to memory accesses. The access time and power consumption of a single access increases with increasing memory size. Therefore, most memory accesses should be made to a small memory. Luckily, most programs exhibit locality, i.e. only a small subset of all variables (=the working set) is accessed frequently. By keeping the working set of the program in a small memory, the memory bottleneck is diminished.**

Traditionally, caches are used which track the working set of a program in hardware. In contrast, we present a compiler method which computes the working set of the program. Based on the analysis, a set of different small memories is constructed tailored to the application. Furthermore, the program is augmented so that it explicitly moves the current working set in the small memories. In comparison to caches, no hardware is needed to keep track of the working set, which makes memory accesses more energy efficient and requires less chip area. After applying this method, execution time was reduced by 17% and energy consumption was reduced by 39% in comparison to the best cache-based solution.

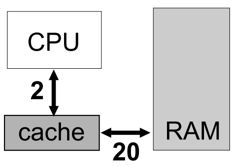
Keywords— **compiler optimization, embedded systems, cache, scratch pad memory**

I. Introduction

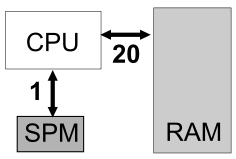
Many programs use a large amount of data, requiring large memories to store it, which degrades efficiency since large memories have long access times and consume a large amount of energy. However, almost all programs exhibit *locality*, i.e. only small subset of all data is frequently used during a given time frame t of program execution. This small subset is called the *working set* of time frame t. By keeping the working set in a small memory, most memory accesses are handled by that small memory, which makes them faster and more energy-efficient.

In most systems, caches are used to make sure that the data in the working set is kept in a small and efficient memory. A *cache* (see fig. 1(a)) is placed between the CPU and the main memory. It consists of a small memory structure to remember the most frequently used data. Furthermore, it contains a hardware tag structure, which records its contents and how frequently data is used. This information allows to retain the most frequently used data in the cache. In embedded systems, caches have a number of drawbacks. First, it is unpredictable whether a memory access will be serviced by the cache or by main memory, leading to unpredictable execution time, which is bad in real-time systems. Furthermore, the tag structure makes the cache larger (in chip area), more energy consuming, and slower in comparison to an SRAM memory of the same size. Therefore, in

K. Beyls is with the Department of Electronics and Information Systems, Ghent University (RUG), Gent, Belgium. E-mail: kristof.beyls@elis.UGent.be .



(a) cache-based system

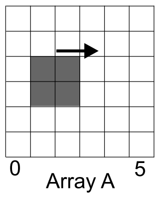


(b) SPM-based system

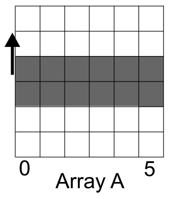
Fig. 1. Difference between a cache-based and an SPM-based system. The number on the arrows show typical relative values of energy consumption cost of a access to a given memory.

```
for i = 0 to 4
  for j = 0 to 4
   B(j,i) = A(i,j)   + A(i+1,j) +
            A(i,j+1) + A(i+1,j+1)
  endfor
endfor
```

(a) example code



(b) j-working set



(c) i-working set

Fig. 2. Example of a code with different stencils. (a) shows the source code, (b) and (c) show the working set in array `A` of a single iteration of the `i`- and `j`-loop, i.e. the data elements accessed by those iterations. Since the `j`-loop is nested in the `i`-loop, the working set of the `j`-loop is a subset of the working set of the `i`-loop.

embedded systems, instead of cache, *scratch pad memories (SPM)* are often used, which are small SRAM memories which are addressed directly by the software (see fig. 1(b)). The difficulty in using SPMs is that the software must control the contents of the SPM by explicitly moving data between SPM and main memory. Here, we propose a program analysis which calculates the working sets of the program at any time during execution. Based on this analysis, it is calculated which size the SPM should be. Furthermore, code is inserted in the application which moves data between SPM and main memory when it enters/leaves the programs working set.

II. Program Analysis

Here, for reasons of clarity and simplicity, the program analysis is explained by example. A more general and for-

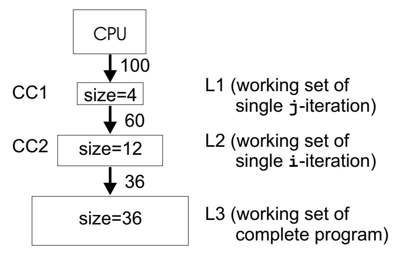

CPU
100
CC1
size=4
60
CC2
size=12
36
size=36
L1 (working set of single j-iteration)
L2 (working set of single i-iteration)
L3 (working set of complete program)

Fig. 3. Copy candidate graph for array `A` in the loop in fig. 2

mal description of the analysis is described in [1]. Consider the code in figure 2(a). For every loop, it is computed which array elements are accessed at a given iteration. The results of this analysis is shown in figure 2(b) for the `j`-loop and in fig. 2(c) for the `i`-loop. The arrows in the figure show how the working set moves through the array in consecutive iterations of the loop. In order to store the elements of the working set of an iteration of the `j`-loop, a SPM of size 4 is needed. Similarly, for capturing the working set of the `i`-loop an SPM of size 12 is needed.

Depending on the number of accesses and the size of the required SPMs, it might not be profitable to use a different SPM for each calculated working set. In fact, each working set is just only a candidate for being copied into a fast SPM memory, hence in this context, they are called *copy candidates*. Since the profitability of assigning a copy candidate (CC) to a SPM is dependent on the size and the number of accesses to a CC, these are computed for all possible CCs and represented in a graph (see fig. 3). Since nested loops form hierarchies, also their working sets and CCs form hierarchies. This is visible in the graph in fig. 3. Consider the copy candidate for loop `i`, of size 4 (CC1). In the graph, this CC is hierarchical higher than the CC2 for loop `j` (size 12), which allows it to fetch all its elements from CC2. Therefore if CC1 is allocated to an SPM of size 4, the CPU will perform 100 requests to that SPM, and 60 memory requests will be needed to lower levels of the hierarchy to copy data into that SPM. However, if CC1 is not allocated to an SPM, all 100 accesses made by the CPU would access the larger and less efficient SPM containing CC2.

Based on the copy candidate graph, it can be decided which CCs should be allocate to which SPM in order to obtain highest efficiency[2]. After it is decided what the best SPM configuration is; the source code is augmented with necessary code which copies data between SPMs and the main memory, in order to keep the working set in the SPMs.

III. Evaluation

The above method has been implemented in a compiler. In order to evaluate the method, the compiler has processed and optimized three typical loop kernels: a matrix multiplication, the SUSAN kernel which performs edge detection and noise filtering in images, and motion estimation which is the most compute-intensive part of video compression. The optimized codes were simulated using Trimaran, and

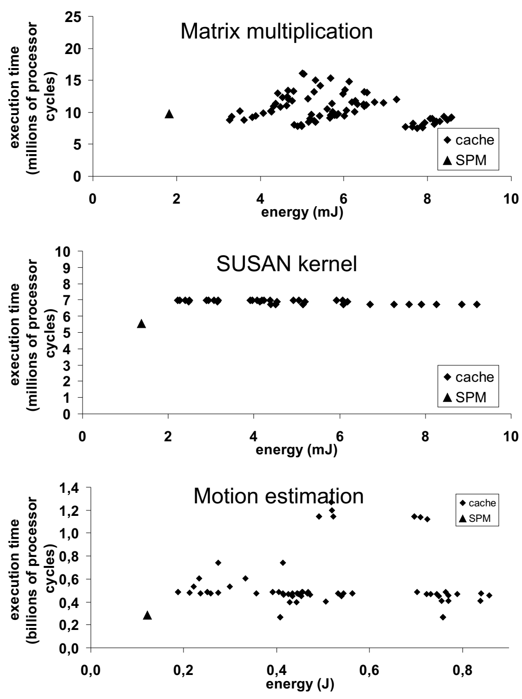


Fig. 4. Results of measuring energy consumption (horizontal) and execution time (vertical) for three programs. The triangle shows the energy consumption and exeution time of the SPM-based solution, while the diamonds show the results for a large amount of different cache configurations.

their energy consumption were computed using the CACTI model. The compiler-generated SPM-based program was compared to solutions using a wide range of caches of different size and associativity. The results are shown in fig. 4. As can be seen, the SPM-based solution is always more energy-efficient, and often faster than the best cache-based solutions.

IV. Conclusions

Moving the responsibility for keeping working sets in small memories from the cache hardware to the compiler software simplifies the hardware and leads to more efficient memory accesses. On our experiments, execution time was reduced by 17% and energy consumption was reduced by 39% in comparison to the best cache-based solution. More details canbe found in[1].

Acknowledgments

This research was supported by the Flemish Institute for promotion of scientific and technological research in the industry (IWT).

References

[1] Kristof Beyls and Erik D'Hollander, "Compiler-generated dynamic scratch pad memory management," in *Workshop on Application Specific Processors*, 2003, submitted.

[2] E. Brockmeyer, M. Miranda, H. Corporaal, and F. Catthoor, "Layer assignment techniques for low energy in multi-layered memory organisations," in *DATE*, 2003, pp. 1070–1075.