Optimising data assignment for energy efficiency on a multi-processor platform

A.J. van der Vegt E. Brockmeyer
IMEC, Kapeldreef 75, Leuven, Belgium
{vdvegt,brockmey}@imec.be

Abstract

Traditional processor design means creating faster processors at ever higher power costs. As power consumption is a key issue in portable applications, this leads to major design problems. Fortunately, multi-processor environments might be the way out. However, little is known on reusing data copies to reduce memory accesses in multi-processor environments. To explore energy-timing trade-offs, a highlevel multi-processor simulator was created and an in depth study on a data dominant application was performed.

1 Problem statement

Existing platforms always have more than one layer of memory hierarchy to bridge the enormous performance and energy consumption gap between the processor and the main memory. We believe a lot of energy can be saved distributing the data well over the memory hierarchy at compile time. In order to control this, we prefer scratch pad memory over caches.

To make use of this intermediate memory level, an application's memory usage is examined closely beforehand. It then can be adjusted to exploit this memory structure to reduce the number of accesses to the highest layer, thus preserving energy. Test results on a single-processor have shown a reduction of up to 50% in the number of main memory accesses using the approach in [1]. When using this technique for multi-processor applications duplicate copies might have to be introduced, and synchronisation becomes a major item. FIFO data-structures make synchronising less problematic, but of course requires space in the memory layers close to the processor. Because of this the results for multi-processor environments will be quite different.

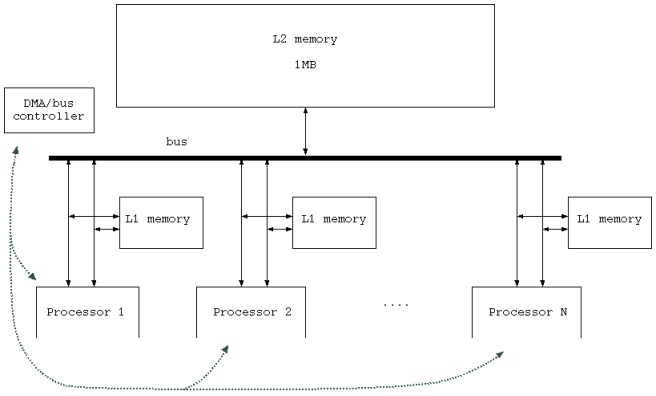

Figure 1: An overview of the simulated multi-processor environment.

In our multi-processor simulator we have investigated two levels of memory, called level 1 (L1), the small memory, and level 2 (L2), the main memory. A simple overview of the multi-processor

environment that has been simulated is shown in figure 1. The simulator itself is transaction based and implemented using Tipsy [2]. We assume all L1 memories to be private (meaning a processor can only access its own L1 memory), and the simulator allows us to program a data controller to copy data from one memory to another.

Often, a small portion of a large data-structure (e.g. an image) is read multiple times. To have this smaller portion of the image put in L1 memory, we add new data-structures to our program. Just before the image is accessed often, part of it is copied to the newly created data-structures, and these are used instead of the larger version. We call this 'data reuse', and those copies 'copy candidates' (CCs). A lot of CCs exist, but only a few should be selected to get the lowest energy usage. Which CCs to select is a key factor in the whole exploration.

Using our simulator we studied an inter-frame compression technique for video images called Quadtree Structured Difference Pulse Code Modulation (QSDPCM). This algorithm involves hierarchical motion estimation, quadtree based encoding of the motion compensated frame, and finally a Huffmann based compession [3]. Video algorithms like this are both regular and very data dominated, which results in characteristics that are very well suited to be exploited using data level parallelism.

Parallelising an application can be done in different ways. Dealing with video images it is easy to simply have each processor deal with its own image. Parallelising in such a way requires large buffers and causes large, unacceptable delays. To avoid this, we split the application on the functional level: first the algorithm performs image manipulation including motion estimation. In the next phase it generates a bitstream. As the motion estimation part takes most of the computation time, we split that part again. We implemented this by dividing the images into rows. Each process deals with its own set of rows, thus exploiting data level parallelism.

Some arrays, CCs and threads of the parallelised QSDPCM algorithm are shown in figure 1. In the simulator, every thread is mapped onto its own processor. Different mappings will be explored in future.

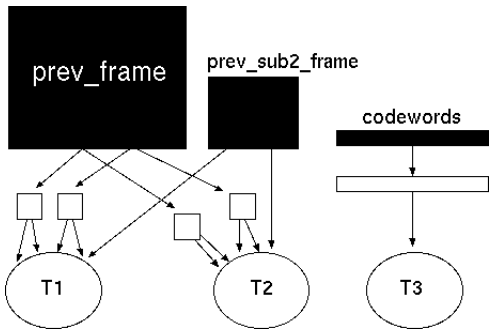

Figure 2: Some QSDPCM copy candidates. The white squares represent CCs; T1, T2 and T3 denote different threads.

2 Exploration methodology

Optimising a program the way we did is a four step process. In the first step, data assignment and data usage information is gathered. This information is used in the second step to explore all data reuse possibilities, taking data lifetime and platform-constraints into account. In the next step, the result of this exploration is used to generate source code. Using this new code the simulator can generate the desired energy and timing results in the fourth step.

Step one is fully automated using the Atomium [4] analysis tool. Atomium instruments the source code, and the access count is generated per array. The second step is partially automated using a MHLA prototype [1]. Given a program and its reuse information, the tool determines which CCs are to be used, and it does so in reasonable CPU time (the tool runtime is a couple of seconds). The third step was partially done by hand using the results produced by MHLA. For our exploration, the MHLA reuse information and platform descriptions had to be modified for multiple processors. For

easier exploration, all CCs are implemented in source code in such a way that preprocessing directives control which copies are activated. These directives were automatically generated from the MHLA output.

A script was made to automate the whole process from initial source code without CCs to a compiled executable with the right CCs implemented. Using this script we have explored over the number of processors simulated and the L1 size. This resulted in energy consumption and execution time estimations.

3 Results and interpretation

Preliminary results are shown in figure 3. The numbers next to the data points indicate the number of processors used. In all cases, one processor is exclusively used by the Huffmann thread creating the output bitstream. This leaves $n - 1$ processors for the QSDPCM algorithm. It can be seen that more

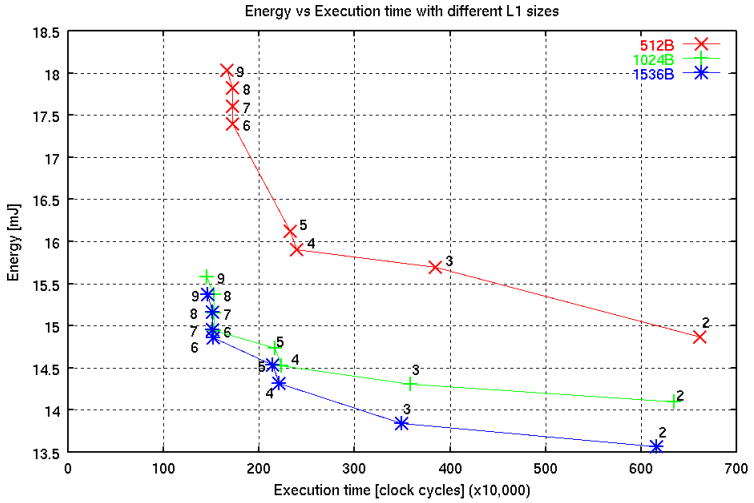

Figure 3: Preliminary results. Time - energy trade-offs exploring the number of processors and L1 size.

processors result in less execution time, but also increase energy usage. For example increasing the number of processors from 2 to 9 decreases execution time by about 75%. At the same time, energy usage rises by about 15%. Varying each processor's L1 size has a significant impact on the energy usage, while its impact on the performance is limited. This is all with a near optimal mapping. As the number of processors increases, steps in execution time are observed. For example, the program takes equally long whether 5, 6, 7 or 8 processors are being used by the QSDPCM algorithm. This stepping behaviour can be explained by the unbalance of the thread load.

References

[1] E.Brockmeyer, M.Miranda, F.Catthoor, H.Corporaal, "Layer Assignment Techniques for Low Power in Multi-layered Memory Organisations", *Proc. 6th ACM/IEEE Design and Test in Europe Conf.* (DATE), Munich, Germany, pp.1070-1075, March 2003.

[2] Tipsy homepage: http://www.imec.be/design/background/tipsy/.

[3] P.Stroback. Qsdpcm - a new technique in scene adaptive coding. In Proc. 4th Eur. Signal Processing Conf. (EU-SIPCO), pages 1141-1144, Grenoble, France, Sept 1988.

[4] Atomium homepage: http://www.imec.be/design/atomium/.