Performance and Power Analysis of Parallelized Implementations on an MPCore Multiprocessor Platform

H. Blume, J. v. Livonius, L. Rotenberg, T. G. Noll
Chair for Electrical Engineering and Computer Systems
RWTH Aachen University
Schinkelstraße 2, 52062 Aachen, Germany
{blume,livonius,rotenberg,tgn}@eecs.rwth-aachen.de

H. Bothe, J. Brakensiek
Nokia Research Center
Meesmannstr. 103,
44807 Bochum, Germany
{harald.bothe,jorg.brakensiek}@nokia.com

Abstract — **In this contribution, the potential of parallelized software that implements algorithms of digital signal processing on a multicore processor platform is analyzed. For this purpose various digital signal processing tasks have been implemented on a prototyping platform i.e. an ARM MPCore featuring four ARM11 processor cores. In order to analyze the effect of parallelization on the resulting performance-power ratio, influencing parameters like e.g. the number of issued program threads have been studied. For parallelization issues the OpenMP programming model has been used which can be efficiently applied on C-level. In order to elaborate power efficient code also a functional and instruction level power model of the MPCore has been derived which features a high estimation accuracy. Using this power model and exploiting the capabilities of OpenMP a variety of exemplary tasks could be efficiently parallelized. The general efficiency potential of parallelization for multiprocessor architectures can be assembled.**

Keywords — Multicore Processors, Parallelization, Power Estimation and Optimization

I. Introduction

In the last decades there have been various approaches in order to increase the computational power of processor architectures. Besides purely increasing the achievable clock frequency by improving the underlying CMOS technologies there have been also various architectural approaches like e.g. the design of application specific instruction set processors [1].

But especially in the field of mobile applications there is not only an increasing need for computational power and flexibility through high level programmability but also the need for high power efficiency. In the last years the increase in computational power has drastically outperformed the increase in battery capacities. Therefore, power efficient processor architectures for mobile applications are the focus of many research projects. For example, it has to be inspected if multiprocessor architectures are a viable option for this [2]. Principally, many tasks in the field of digital signal processing feature a high degree of inherent parallelism. But it has to be discussed what effort is related to the problem of parallelizing processor code. This extra effort concerns the programming work in the design phase of such code as well as the power efficient handling of such parallelized applications on a multicore architecture.

Therefore, a variety of typical digital signal processing tasks have been implemented on a prototyping platform featuring a high performance ARM MPCore with four ARM11 processor cores. The inspected typical signal processing tasks range from basic FIR filtering tasks or block matching for motion estimation purposes to complete JPEG2000 en/decoders or encryption algorithms. All of these algorithms have been implemented on the processor architecture in serialized as well as parallelized versions. Several parallelization parameters like the number of issued threads or the granularity of parallelization have been varied and analyzed. The parallelization has been implemented on a high abstraction level using the OpenMP programming model. OpenMP is an open specification which allows to efficiently parallelize C/C++ -programs for parallel processors featuring a shared memory (Fig. 1) by adding specific OpenMP directives into the C-program code. These directives support the distribution of autonomous subtasks (threads) over the available processor cores. This programming model allows for example to incrementally parallelize program code.

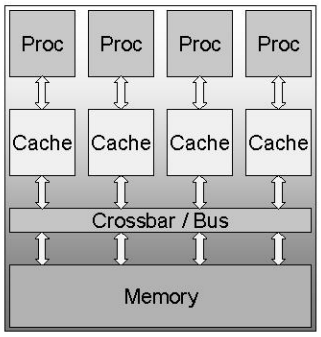

Figure 1. Shared memory model multiprocessor architecture

In order to (power) optimize the parallel implementation of tasks on this processor architecture an estimation of the related power consumption of a given implementation is advantageous. This releases the programmer from performing the task-under-test on a prototyping platform and from measuring the related power consumption of each new software optimization level.

1-4244-1058-4/07/$25.00 ©2007 IEEE

The availability of an accurate power model allows to efficiently navigate in the design space of digital signal processing on multicore architectures. Therefore, such a power model for the ARM MPCore architecture has been derived. The power model is based on the concept of so-called hybrid functional level/instruction level power analysis (FLPA/ILPA) [3] which has been successfully applied to various processor architectures before. The main steps of deriving this model for the MPCore and the achievable estimation accuracy will be discussed.

On the basis of OpenMP and the power estimation model the exemplary parallelization and optimization of the signal processing tasks is performed. Comparing the results for various implementations helps to explore the available potential of parallelization and to understand which influence specific parallelization parameters have.

The paper is organized as follows: Chapter II shortly discusses the basic architecture of the MPCore architecture and the MPCore prototyping platform used in the course of this work. The following chapter describes the fundamentals of parallelization of software on a multiprocessor platform using OpenMP. Chapter IV discusses the parallelization of an exemplary block matching algorithm on this architecture. Chapter V works out the hybrid functional level/instruction level power model of the MPCore. A performance and power benchmarking of parallelized implementations is performed in chapter VI. Finally, a conclusion is given in chapter VII.

II. MPCore Architecture

The ARM MPCore [4] is a synthesizable multiprocessor implementing the ARM11 micro architecture. Here, only those basics of the MPCore architecture are briefly sketched which are required for its power modeling and for understanding efficient parallelization strategies on that processor. According to the MPCore concept, this multiprocessor can be configured by customers during the design phase. Generally, the MPCore provides the following (configurable) features:

- 1 to 4 ARM11 processors,
- high performance memory system,
  - L1 data and instruction cache per processor from 16 KByte to 64 KByte,
  - a snoop control unit (SCU) that connects the ARM11 CPUs to the memory system featuring inter CPU communication (Direct Data Intervention (DDI)),
- single or dual 64-bit AMBA 3 AXI bus,
- optional Vector Floating Point unit (VFP),
- up to 255 hardware interrupts.

The ARM11 cores of the MPCore are based on the ARMv6K architecture. Support for the ARM Thumb instruction set (16 bit instructions) is provided and the Jazelle extension (for Java bytecode execution) as well as DSP and SIMD ISA extensions are included. Furthermore, a high energy efficiency is targeted by implementing so-called Intelligent Energy Management features (IEM) which include a shutdown of unused resources (for each processor independently) and a dynamic voltage and frequency scaling support.

The MPCore possesses a high performance memory system with a multi-level cache hierarchy and features separate data and instruction caches per CPU and the possibility to easily shift data between the single caches.

For the course of this work a prototyping MPCore chip manufactured in a 130 nm CMOS technology resulting in an adaptable CPU clock frequency of up to 300 MHz has been applied. This prototyping chip features four modified ARM11 CPUs incl. a Vector Floating Point Unit. It provides a L1 memory subsystem (per CPU) with 32 KByte instruction cache and 32 KByte data cache. The L2 memory subsystem provides 1 MByte L2 unified cache. Hence, the MPCore follows the shared memory paradigm (see Fig. 1) which is required for the parallelization of tasks on a multicore processor with OpenMP. Fig. 2 depicts a block diagram of the MPCore architecture used here.

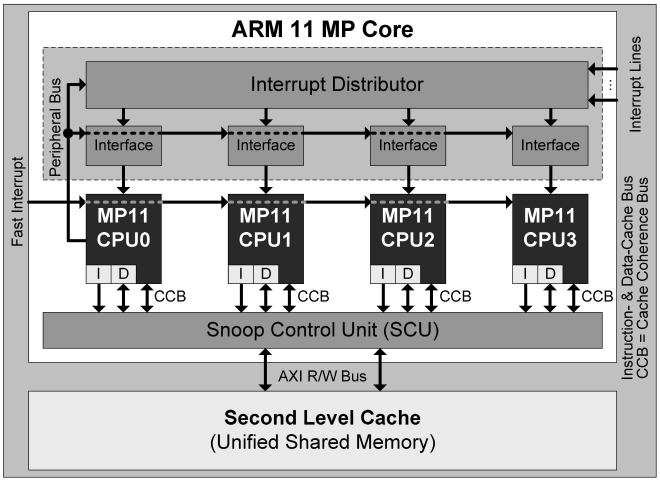

Figure 2. MPCore testchip, block diagram

Each ARM11 CPU can be configured to one of the following modes:

- Run mode: everything is clocked and powered-up,
- WFI (Wait For Interrupt) mode: CPU clock is stopped, only logic needed for wake-up is still active,
- Dormant mode: Everything is powered off except RAM arrays that are in content retention mode,
- Powered-off.

The processor core provides eight pipeline stages (two fetch-, a decode-, an issue- and four integer-execution stages). Static as well as dynamic branch prediction is applied.

The Vector Floating Point Unit (ARM VFPv2 denoted here as VFP11) provides a low power consumption and is optimized for a high data rate as well as a fast and parallel execution of division/square root and further arithmetic operations. This parallel execution is achieved by implementing three separate instruction pipelines. The VFP11 features a

- Multiply and Accumulate (FMAC) pipeline,
- Division/Square root (DS) pipeline,
- Load and Store (LS) pipeline.

Each pipeline is working separately from each other. The FMAC and LS pipeline provide a single cycle execution.

Fig. 3 depicts the prototyping platform (Versatile Emulation Baseboard with ARM11 MPCore Core Tile [5]) applied here which features besides the MPCore an FPGA realizing the configuration interface as well as the peripheral controller, 128MByte DDR SDRAM, 2MByte SRAM, 64MByte NOR flash and various external interfaces.

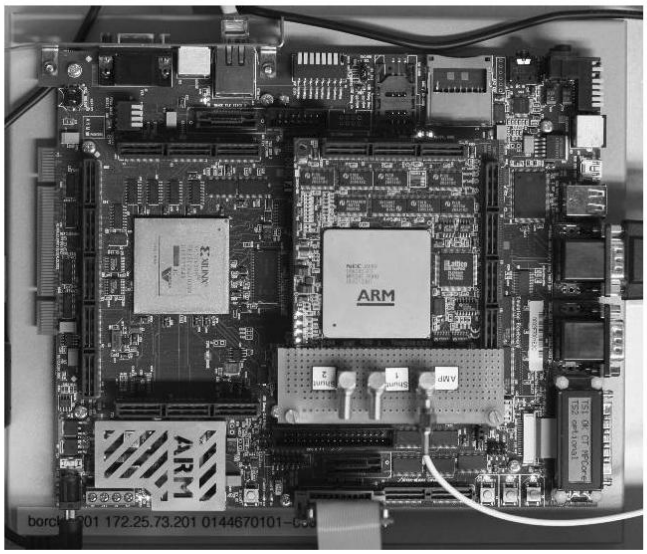

Figure 3. ARM MPCore prototyping platform

III. Parallelization of Programs using OpenMP

In contrast to the programming of sequential tasks on a single core architecture, on a multicore architecture the programmer has to decide how the work should be distributed across multiple processors. Actually, the POSIX thread library is often used to develop parallelized code. Alternatively, this additional development step can be realized using the parallel programming model of OpenMP for shared memory multiprocessors [6]. It provides the advantage to simplify managing and synchronization of program threads. OpenMP works in conjunction with the prevalent programming languages Fortran and C/C++. Therefore, a set of compiler directives that control the distribution of tasks over the processor cores and the necessary synchronization of these tasks, are available. Additionally, a supporting library of subroutines is provided. The OpenMP API is independent of the used platform and operating system. Appropriate compilers exist for a variety of all major operating systems. Hence, porting OpenMP programs is in many cases only a matter of recompiling.

The directives are instructional notes to any compiler supporting OpenMP (e.g. GCC 4.2). To enhance application portability they take in case of C/C++ programs the form of *#pragmas*, so they will be ignored by any compiler not supporting OpenMP. This directive-based parallelization approach has the benefit that it allows the same source code to be used for single- and multiprocessor development, since the code will be executed serially on single core and in parallel on multicore processors. Furthermore, it allows an incremental parallelization approach starting from an existing serial version by adding parallel code regions step by step.

The OpenMP language extensions can be separated into control structures for expressing parallelism and work-sharing on the one hand and data environment constructs for inter-thread communication and synchronization on the other. Fig. 4 gives an overview of the most important OpenMP directives. The control structures for parallelization (i.e. *parallel*) are embedded into a so called fork/join execution model. Thus, they fork (i.e. start) new threads and execute an enclosed code block concurrently, and afterwards they join in parallel running threads to a serial master thread. By means of work-sharing directives the work within a code block can be divided among such an existing team of threads. An instance for this is the *for* directive, which divides loop iterations among concurrently executing threads, and therefore exploits the loop-level parallelism. The required thread synchronization can be done implicitly by OpenMP e.g. at the end of a parallel region (join) or explicitly by the programmer through directives like *barrier* (wait until barrier is reached by all threads) or *critical* (exclusive access of code regions).

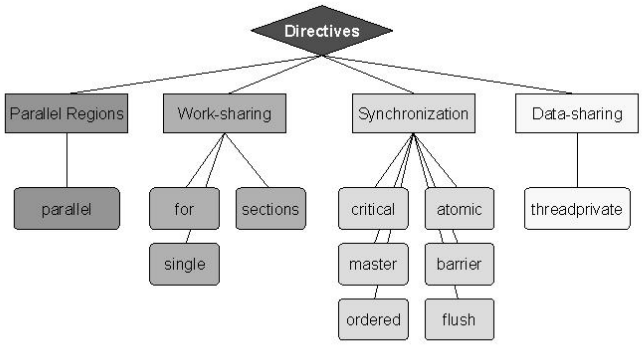

Figure 4. Overview of main OpenMP directives

In Fig. 5 the parallelization of a for-loop is given as an example. Additionally, the fork/join principle and the work distribution on a team of four in parallel running threads are visualized.

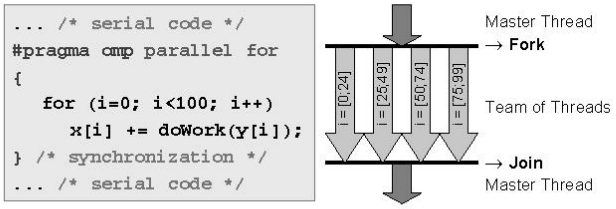

Figure 5. Fork-join principle using a basic OpenMP example

IV. Exemplary Parallelization of a Motion Estimation Algorithm

The parallelization with OpenMP on the MPCore multiprocessor platform was applied to several signal processing algorithms of varying complexity. Here, the exemplary parallelization of a block matching motion estimation algorithm will be presented. Motion estimation is a basic building block of video processing systems like motion compensated filtering, image coding or motion vector based interpolation for video format conversion [7].

The most important method for motion estimation is block matching. Here, every image of a sequence is divided into blocks of equal size. For each block the position with the highest correlation of the image content is searched in an

adjacent image of the sequence. The displacement vectors between those block positions represent the resulting motion vectors. The block matching algorithm with the most regular structure is the full-search algorithm, which uses a brute-force strategy (i.e. testing for every block all possible positions within a search window) to find the best fitting block in an adjacent frame. Hence, the calculation of this block based correlation function represents the core piece of the algorithm and is embedded within several nested loops (Fig. 6).

```
#pragma cmp parallel for (...)
{
for (by = 0; by < numBlocksY; by++)
   for (bx = 0; bx < numBlocksX; bx++)
      for (sy = 0; sy < searchWindowY; sy++)
         for (sx = 0; sx < searchWindowX; sx++)
            calcCorrelation(bx, by, sx, sy,...);
}
```

Figure 6. Exemplary parallelization of the full-search block matching loops

Due to the high regularity and weak data dependencies the parallelization of this algorithm with OpenMP is straight forward and can be reduced to a single additional code line. Nevertheless, the programmer has to decide which for-loop should be parallelized. Here, it is best to select the outermost loop, which controls the vertical iteration over the search blocks, since there are no data dependencies between the correlation calculations of the single blocks. Due to this, the workload being distributed over simultaneously running threads can be maximized and the synchronization overhead can be minimized.

In Fig. 7 the impact of the parallelization with OpenMP on speedup, power consumption and the resulting relative efficiency (see definition of efficiency in equ. (4)) is depicted.

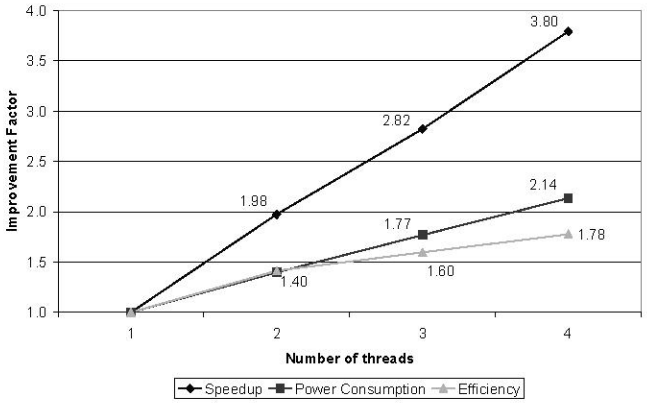

Figure 7. Exemplary parallelization of a block matching algorithm

For this analysis, the corresponding power consumption of each code version has been directly measured at the MPCore. All results were normalized with regard to the values measured for a single thread. As can be seen, the achieved speedup scales nearly perfectly linearly. The power consumption increases with a significantly lower slope than the speedup, and therefore the relative efficiency increases constantly over the number of threads.

V. HYBRID FLPA/ILPA MODELING OF THE MPCORE

In order to perform energy-aware optimization of the program code, a power model of the MPCore processor has been elaborated which allows prediction of the according power consumption of code that is executed on the processor. Hence, the so-called hybrid functional level/instruction level power analysis (FLPA/ILPA) technique has been applied in order to derive such a power model.

According to this methodology, in a first step the processor architecture is divided into functional blocks like the processing unit, the internal memory and others like the clocking system. While executing specific test scenarios on the processor and by performing simulations or measurements it is possible to find an arithmetic model for each block that determines its power consumption dependent on certain parameters. These parameters are, for example, the degree of parallelism, the access rate of the internal memory or the clock frequency. Most of these parameters can be automatically determined by a parser which analyzes the assembler code. Further parameters can be derived from a single execution of the program (e.g. the number of required clock cycles). These parameters are the input values for the previously determined arithmetic models. Thus, an estimation of the power consumption of a given task can be computed.

If the processor features a strong dependency of the power consumption on the currently executed instruction the pure functional model has to be extended by instruction dependent elements. According to this approach, the ISA of the processor is classified into several instruction classes with according arithmetic power functions leading to a hybrid FLPA/ILPA model. Such models have been successfully elaborated for a variety of processors (see e.g. examples in [3], [8]).

The inspection of the instruction specific power consumption of one ARM11 core of the MPCore resulted in a power model where three separate instruction classes, i.e.

- arithmetic operations (ADD, CMP,...) incl. as well single and multiple store operations (STR, STM,...),
- single and multiple load operations (LDR, LDM,...),
- exclusive memory accesses (LDREX, STREX, SWP,...),

respectively three according power functions have been derived.

Due to the higher power dynamics between the single VFP11 instructions, for the VFP11 a more differentiated model featuring five instruction classes respectively power functions has been elaborated, i.e.

- basic data processing of the FMAC pipeline and data transport between ARM11 and VFP11 registers (FADDD, FABSD, FMRS,...),
- complex data processing of the FMAC pipeline (FMACD, FMULD,...),
- instructions of the DS pipeline (FDIVD, FSQRTD,...),
- instructions for the single write memory accesses of the LS pipeline (FSTD, FSTS,...),
- instructions for single read and multiple memory accesses of the LS pipeline (FLDD, FLDMD, FSTMD,...).

Fig. 8 and Fig. 9 depict the derived power functions P_{instr_spec} for the ARM11 core and for the VFP11.

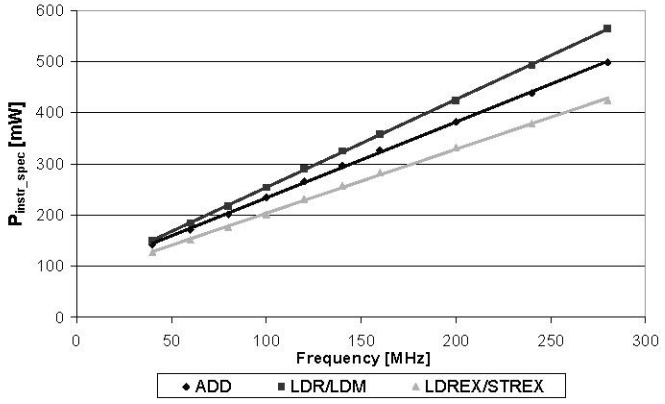

Figure 8. The instruction-specific power consumption for the three ARM11 MPCore CPU instruction classes

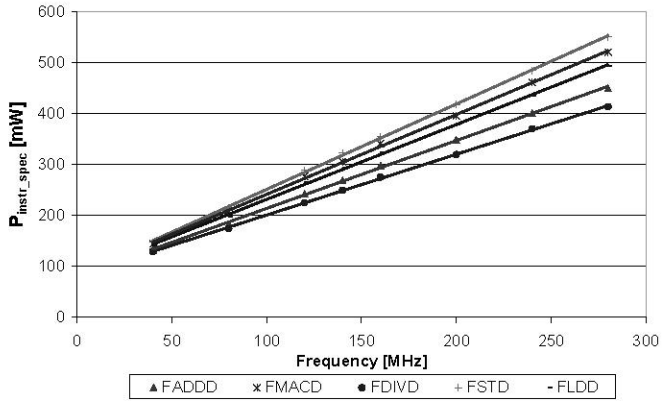

Figure 9. The instruction-specific power consumption for the five VFP11 coprocessor instruction classes

Besides the consideration of instruction dependent power consumption also the influence of the memory system i.e. cache misses has to be regarded. Therefore, the number of instructions in the applied test scenarios (here more than 8192 instructions, due to the cache size of 32KByte and the instruction and data word length) has been successively increased to enforce different numbers of cache misses. The corresponding power consumption was no longer a linear function of the core clock frequency while the processor was executing those test scenarios.

The difference at a given frequency between the instruction-specific power consumption P_{instr_spec} and the actual power consumption executing such test scenarios with cache misses is called the bus-specific offset P_{bus_spec}. Hence, the number of cache misses would be an appropriate parameter influencing the model for P_{bus_spec}. Using the ARM instruction set simulator and cycle counter it is possible to derive various cycle counts (core clocks, memory bus clocks, etc.). These values are much more accurate than the number of cache misses which are also provided by the simulation environment [4], [9]. Therefore, the bus-specific offset can be modeled as a linear function of the ratio $1/T$. Here, T denotes the total number of memory bus cycles and can be determined either with the RVDS-simulator or with the ARMulator [10]. Besides the dependency on the ratio $1/T$ the bus-specific offset is also a function of the frequency. It can be modeled as

$$P_{bus\_spec}(f,1/T) = a \cdot f + b \cdot 1/T + c \cdot f \cdot 1/T + d \tag{1}$$

where negative values for $P_{bus\_spec}(f,1/T)$ are not possible and therefore clipped to zero.

Finally, it is possible to calculate the actual power consumption of an instruction specific test scenario by

$$P_{act}(f,1/T) = P_{instr\_spec}(f) - P_{bus\_spec}(f,1/T) \quad . \tag{2}$$

To estimate the complete power consumption of the MPCore processor while executing a complex task, a profiler from the RVDS framework [9] determines the share h_{label} of the execution time of the different parts of the assembler code which are produced by the compiler and which are denoted here as labels. The instruction distribution is determined for every label by a special parser which has been implemented as a *C* program, whereby the complete shares h_i of every instruction class i (ARM11 CPU) resp. h_j (VFP11) in the label are extracted. As described before, the parser categorizes the instruction set into three different instruction classes for the CPU instructions and five instruction classes for the VFP11 instructions. Since memory accesses are covered by specific arithmetic power functions (see for example the aforementioned list of instruction classes for the ARM11 CPU) no additional modeling of the inter-CPU communication is required. Therefore, the resulting hybrid FLPA/ILPA power model of the MPCore can be assembled as a sum over all cores inside the MPCore processor

$$P_{MP-Core}(f,1/T) = P_{offset}(f) + \sum_{Cores}\sum_{label} h_{label} \cdot \left(\sum_{\substack{instr\ classes\ i \\ CPU}} h_i \cdot \left(P_{instr\_spec}(i,f) - P_{bus\_spec}(i,f,1/T)\right) + \sum_{\substack{instr\ classes\ j \\ VFP11}} h_j \cdot \left(P_{instr\_spec}(j,f) - P_{bus\_spec}(j,f,1/T)\right) \right) \tag{3}$$

Here, $P_{offset}(f)$ includes the power consumption of the clock network as well as the instruction independent power consumption of all cores when all cores are in the wait for interrupt mode.

The estimated power consumption was compared to the measured values for a variety of tasks out of the field of digital signal processing in order to benchmark the hybrid FLPA/ILPA model. Fig. 10 shows the results of the benchmarking. The comparison of estimated and measured values yields a maximum error of about 6% and an average error of about 3% for the power consumption. As can be seen in Fig. 10, the variety of tasks which has been inspected on this platform features a dynamics concerning the according power consumption of about 22% (e.g. Cache Test: 327 mW, Serial Search: 418 mW). Thus, the estimation error is much smaller than the power consumption dynamics and the model can be used successfully for the estimation.

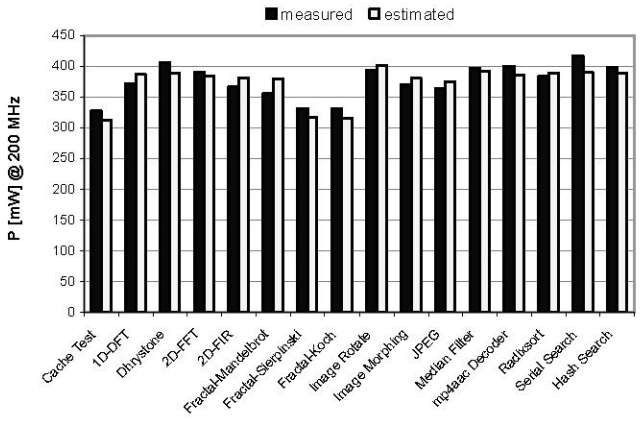

Figure 10. FLPA/ILPA estimation results and measurements for the MPCore architecture

VI. Evaluation of the MPCore Architecture Executing Parallelized Tasks

The hybrid FLPA/ILPA power model has been applied in order to elaborate power efficient serial reference code for a variety of basic tasks of digital signal processing. Additionally, the OpenMP directives have been used in order to exploit the inherent parallelism of the implemented tasks by applying these directives to the initial serial program code. Hence, only marginal programming effort was required for the parallelization. Tab. I describes the main features of the implemented tasks.

In order to quantitatively analyze the influence of different parallelization schemes, a quantitative measure is required. In the following, the efficiency metrics η defined as

$$\eta = \frac{relative\ speedup}{relative\ power} \tag{4}$$

will be used. Here, *relative speedup* denotes the ratio of the throughput of a parallelized code version compared to the throughput of a purely linear code version. Consequently, *relative power* denotes the ratio of the power consumption of a parallelized code version compared to the power consumption of a purely linear code version.

For the following results the number of threads has been fixed to four. In Fig. 11 a) the relative speedup which could be achieved for the parallelized code versions, the relative power consumption (Fig. 11 b) and resulting from this, the relative efficiency gain (Fig. 11 c) are depicted.

It can be seen that by using four threads on the MPCore architecture an average efficiency gain of 1.8 (average speedup 3.4, average power increase 1.9) could be achieved. The highest speedups and efficiency gains can be achieved for highly regular algorithms like block matching (see chapter IV), filtering or image transformations (speedups ≈ 3.7-3.9, efficiency ≈ 1.8-2.2) which feature regular inner loops which can be advantageously parallelized.

TABLE I. Inspected Parallelized Tasks on the MPCore Platform

Task	Description
Block Matching	full search block matching for motion estimation in video sequences
Image Morphing	transformation (warping) of images
Radixsort Algorithms	bit-level sorting algorithms based on the radix exchange (divide-and-conquer) or the straight radix principle
JPEG2000	wavelet-based image compression technique, here: JasPer OpenSource Code for JPEG2000 [11]
1D/2D FIR Filter	one- and two-dimensional FIR filtering of image data (1D:800 taps, 2D:10x10 taps)
Serial Search	searching of strings in text data bases
1D DFT	one-dimensional discrete Fourier transform (256 point DFT)
2D FFT	two-dimensional fast Fourier transform (256x256 point FFT)
Median Filter	5x5 median filter based on the odd-even transposition algorithm
Traveling Salesman	solving the traveling salesman problem (visiting n points within a tour heading for minimization of tour length) by a branch-and-bound-technique
AES Encryption	encryption technique based on the Rijndael algorithm
Huffman Codec	encoding and decoding of bit streams using a Huffman Code-book
Computation of Fractals	computation of a set of fractals (self-similar shapes); here: Mandelbrot, Koch, Sierpinski

Some specific results shall be discussed here. For example, there is a significant difference in the achievable efficiency concerning the Radix Exchange and the Straight Radix sorting algorithm. Both algorithms are bit-level-based sorting algorithms, but the reason for the different performance is that within the Radix Exchange algorithm a divide-and-conquer strategy is applied. From sorting step to sorting step the field of elements which shall be sorted is subdivided and sorted separately. This enables to use separate processor cores for sorting the sub-fields and therefore a good speedup (2.9) is achieved. Radix Exchange is well suited for using 2^n parallel threads. On the other hand, the Straight Radix algorithm provides some parallelization disadvantages. Here, the elements to be sorted are processed bit-level by bit-level which

can't be parallelized. The only processing step that can be parallelized is the counting of ones and zeros inside the processing of each bit-level. Therefore, the achievable speedup (1.4) is significantly lower.

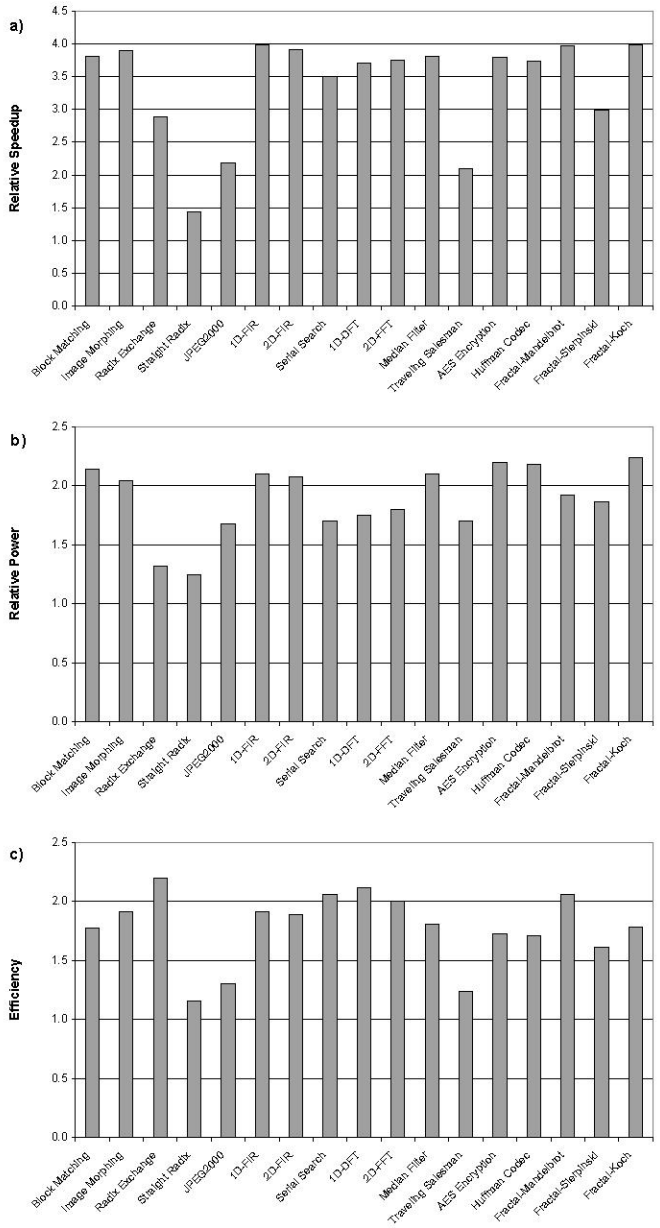

Figure 11. Parallelization results on the MPCore architecture (number of threads is fixed to four) a) relative speedup b) relative power c) efficiency

A further example whose results shall be briefly analyzed is the JPEG2000 encoder. Here, mainly the computation of the discrete wavelet transform (DWT) could be parallelized featuring high speedups. Further elements of this algorithm (e.g. subband processing) can't be effectively parallelized at least on the basis of the JasPer reference implementation [11]. A variety of recursive function calls is included here, which can't be parallelized directly with OpenMP. A better efficiency gain is to be expected if a new JPEG2000 software implementation would be applied which no longer features recursive function calls.

Generally, the parallelization results which could be derived outperform the results of similar inspections on alternative multiprocessor architectures. For example, in [12] OpenMP has been used to parallelize basic operations on a Renesas M32R (dual core) [13] processor. Due to the restricted number of two cores the influence of higher parallelization degrees could not be studied there. Furthermore, the influence on the resulting power consumption has not been inspected.

The programming effort for adding the required OpenMP directives is very small. As a first measure for the required programming overhead the number of additional OpenMP code lines which have to be added to the initial code can be used. In Tab. II these numbers of additional OpenMP code lines for all the inspected tasks are listed. Besides these code modifications only some marginal modifications like e.g. replacing while-loops by for-loops (while loops cannot be parallelized by OpenMP in C/C++) were required.

TABLE II. NUMBER OF ADDITIONAL OPENMP-CODE LINES FOR THE PARALLELIZATION OF THE PROGRAM CODE

Task	# additional OpenMP code lines
Block Matching	1
Image Morphing	1
Radixsort Algorithms	
• Radix Exchange	10
• Straight Radix	2
JPEG2000	3
1D/2D FIR Filter	1
Serial Search	1
1D DFT	4
2D FFT	2
Median Filter	2
Traveling Salesman	8
AES Encryption	7
Huffman Codec	6
Computation of Fractals	
• Mandelbrot	1
• Sierpinski	4
• Koch	5

Up to now only parallelized implementations have been discussed which were achieved by adding a small number of OpenMP directives. Furthermore, it can be discussed which influence parameters like the issued number of program threads or even the use of hand-optimized assembler code will have. Therefore, in Fig. 12 the specific gain (measured in throughput per energy) is depicted over the number of threads for the Huffman Codec (separately for encoder and decoder). Besides the parallelized C version of the encoder and decoder, hand optimized assembler versions were examined too. These assembler versions nearly double the throughput per energy in comparison to the C version in case of four threads. This indicates that the available compiler is not capable of using the full potential of the MPCore.

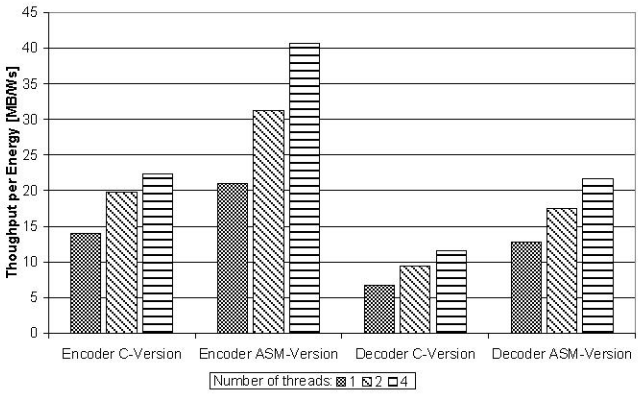

Figure 12. Througput per energy of Huffman encoder and decoder; influence of the number of threads and comparison of C-code and hand-optimized assembler versions

A further example where the influence of the issued number of threads can be studied is the Traveling Salesman Problem. Using four threads (see Fig. 11) only a speedup of 2.1 and an efficiency gain of 1.2 could be achieved. It has been also studied what factors can be achieved if the issued number of threads is greater than the number of available processor cores. Using for example eight threads further increases the speedup to a factor of 2.6 and issuing 16 threads results in a speedup of 2.9. Within the single threads it may occur that their processing time differs significantly. In such a case, a single thread which requires a long computation time can block the further processing of the complete task. Therefore, it is often beneficial to increase the number of parallel threads. If one thread is still allocating a core, the next threads can be distributed over the available cores (i.e. using a dynamic scheduling scheme for the issued threads). OpenMP here also provides the possibilities to easily modify this parameter number of threads (e.g. as an attribute of the OpenMP directive *parallel*). This example also shows how the software designer can effectively play around in the implementation design space of parallel processor architectures.

Generally, the discussed examples show that for a variety of tasks which provide an inherent degree of parallelism significant efficiency gains can be achieved by only very moderate programming effort for parallelizing these tasks i.e. using OpenMP directives. Using hand-optimization of the program code i.e. using assembler coded programs will typically further increase the throughput and the efficiency of the implementations.

The experience from the variety of our test implementations suggests that parallelization with OpenMP results in a very attractive performance-effort relation which significantly outperforms serialized implementations and which often does not need to be improved by assembler-based hand-optimization.

VII. Conclusion

Various typical tasks out of the field of digital signal processing have been implemented in serialized as well as parallelized code versions on a modern ARM11 MPCore multiprocessor platform. For parallelization purposes the OpenMP programming model has been efficiently applied. Typical speedup, power and efficiency numbers could be derived on that platform. Applying a power model for that multiprocessor that has been also elaborated in the course of this work, all code versions could be efficiently power-optimized even in the development phase.

It can be shown that by investing only a very moderate programming effort an average speedup of a factor of 3.4 and an average efficiency gain of a factor of 1.8 can be achieved. Furthermore, the influence of some specific factors like the number of issued program threads or the use of assembler-coding has been inspected.

Our results suggest that very attractive performance-effort ratios can be achieved by OpenMP-based high-level language parallelization on modern symmetric multiprocessor platforms like the ARM MPCore. Hence, the inherent parallelism of signal processing algorithms and the parallel computation capabilities of such kind of symmetric multiprocessor architectures can be efficiently exploited in the future.

References

[1] M. Gries, K. Keutzer, H. Meyr, G. Martin, Building ASIPs, Springer 2005

[2] W. Wolf, "The Future of Multiprocessor Systems-on-Chips, " Design Automation Conference, 41st Conference on (DAC'04), 2004, pp. 681-685

[3] H. Blume, D. Becker, L. Rotenberg, M. Botteck, J. Brakensiek, T. G. Noll, "Hybrid Functional- and Instruction-Level Power Modeling for Embedded and Heterogeneous Processor Architectures," invited paper for the Journal of System Architectures, 2007

[4] "ARM11 MPCore Processor Technical Reference Manual", ARM Limited, Lit.-Nr.: ARM DDI 0360D, 2006

[5] "Core Tile for ARM11 MPCore User Guide", Ref: DUI 0318C, 2006

[6] R. Chandra, Parallel Programming in OpenMP, Morgan Kaufmann, 2001

[7] G. de Haan, Video Processing for Multimedia Systems, University Press, Eindhoven, 2000

[8] H. Blume, M. Schneider, T. G. Noll, "Power Estimation on a Functional Level for Programmable Processors," Proc. of the TI Developers Conference 2004, Houston, Texas, February 2004

[9] "RealView Debugger Version 1.8 Extensions User Guide", ARM Limited, Lit.-Nr.: ARM DUI 0174G, 2005

[10] "RealView ARMulator ISS Version 1.4 User Guide", ARM Limited, Lit.-Nr.: ARM DUI 0207C, 2004

[11] JPEG2000, JasPer Project Homepage, http://www.ece.uvic.ca/~mdadams/jasper

[12] Y. Hotta, M. Sato, Y. Nakajima, Y. Ojima, "OpenMP Implementation and Performance on Embedded Renesas M32R Chip Multiprocessor," Proceedings of the 6th European Workshop on OpenMP (EWOMP 2004), Stockholm, Oct. 2004

[13] S. Kaneko, "A 600 MHz Single-Chip Multiprocessor with 4.8 GB/s Internal Shared Pipeline Bus and 512kB Internal Memory," Proceedings of the ISSCC 2003, Vol. 1, pp. 254-255