Exploiting Fine- and Coarse-grain Parallelism in Embedded Programs

Ireneusz Karkowski and Henk Corporaal
Delft University Of Technology
Information Technology And Systems
Mekelweg 4, 2628 Cd Delft, The Netherlands
I.Karkowski@its.tudelft.nl, heco@cardit.its.tudelft.nl

Abstract

Due to the technological advances, mapping of embedded applications onto single-chip multi-processor systems becomes a feasible and very interesting option. What is needed is an environment that supports the designer in transforming an algorithmic specification into a suitable parallel implementation. In this paper we present the results of our experiments with one such an environment, which we developed within our laboratory. As opposed to the existing ones, our framework semi-automatically exploits different kinds of coarse and fine-grain parallelism from an embedded program written in ANSI C. It employs functional pipelining [12] and data set partitioning [17] simultaneously with source-to-source program transformations to obtain the most advantageous hierarchical parallelizations. This combination results in high speedups for all tested benchmarks.

1. Introduction

As the application area of the embedded processors widens, the demands on their performance are constantly growing. Until now, instruction level parallelism has been successfully exploited to satisfy these high performance requirements. Practice shows however that increasing the number of concurrently operating functional units of typical ILP (instruction level parallel) architectures above a certain level does not necessarily lead to significant performance gains [9]. Instead, high hardware costs and inefficient use of this hardware occurs. The advent of sub-micron processing, allowing integration of millions of transistors on a single carrier, has brought new opportunities in the embedded system design. A multi-processor embedded system becomes nowadays a very interesting alternative. This both in terms of the hardware cost and performance. Especially, if the system consists of several (different) ASIPs (application specific instruction set processor), each with functionality optimized for the subtasks which they have to perform. Code partitioning among the processors leads then to exploitation of the course-grain parallelism (task parallelism and parallelism in loops [5]), while the fine-grain (instruction level) parallelism [9] is exploited locally by each of the processors.

The design of embedded systems is a very tedious process, requiring simultaneous consideration of software and hardware components. To ease these tasks in the past several environments for the embedded system design have been realized (a lot of references can be found in [7, 8, 11]). Also a number of papers specifically about multi-processor system design have been published [3, 2, 16]. However, none of them addresses the problem of the automatic extraction of the parallelism from the system specification.

In this paper we present the first results obtained with our system for multi-processor embedded system design. The system was designed for mapping of an embedded application written in ANSI C onto a cost-efficient heterogeneous multi-processors. Uniqueness of our approach lies in a combination of the state of the art automatic ASIP synthesis software with a coarse- and fine-grain parallelism exploitation methodology.

The paper is organized as follows. In section 2 we introduce the problem of mapping of specific applications onto multi-processors. After that, section 3 discusses different parallelization methods. Our system design space exploitation algorithm is presented in section 4. The performance of the algorithm is demonstrated and evaluated on a number of DSP benchmarks in section 5. Section 6 concludes the paper.

2. Problem statement

In this section we discuss the problem of mapping specific applications onto a multi-processing embedded systems. For this discussion, and also in the remainder of the paper, we use the following example:

Example 1 (Instant frequency tracking)
Figure 1 presents the simplified ANSI C source code of the frequency tracking embedded system from [6]. In its main loop, the program reads a stream of samples and uses LMS (adaptive signal enhancement) to determine instantaneous frequency estimates. A 1024-point FFT (fast Fourier

transform) is then used to determine the frequency response of the adaptive filter every 100 input samples. The procedures **lms** *and* **fft** *are quite complex them self and contain a number of nested* `for` *loops.*

```
float b[128];
COMPLEX samp[1024];

while (forever) {
    for (i=0; i<100; i++) {
        x=getinput();
        lms(x,oldx,b);
        oldx=x;
    }
    Copy b to samp[0..127];
    fft(samp);
    Determine the frequency f from samp;
    sendout(f);
}
```

Figure 1: The simplified source code of the instant frequency tracking application.

Design of an embedded system, like the one presented above, involves finding a mapping Γ of a program graph $G_P(V, D, E)$ onto an architecture template graph $G_A(P, M, C)$, such that the resulting hardware-software solution satisfies the price-performance specification for the design. In these graphs, V is the set of program statements, D the set of data objects, E the set of data and control dependencies between the statements, P the set of processing elements, M the set of memories, and C the interconnection network between them. An example multi-processor architecture can include a set of ASIPs, possibly with local memories, communicating via a combination of fast interprocessor links and/or shared memory. In our case the ASIPs are designed using the MOVE framework [5]. The architectural variables defining the design space are shown in table 1. Our goal is to find a design trade-offs curve Π, connecting a set of Pareto points in the 2-dimensional cost-performance space [10] for a given application (as the cost usually the chip area, power dissipation or number of packaging pins is used). As example, figure 3 shows the design trade-offs curve for the system implementing application from example 1. Having it, we can select a cost-efficient and feasible (satisfying the constraints) solution. As an example, consider again the source code from figure 1.

Example 2 (A mapping)
The main loop body can be mapped onto a 2-processor computer system, in which the first loop (with the **lms** *function inside) is mapped onto the first processor in the chain, and the remaining code, including the* **fft** *onto the second one. Clearly when we partition data dependent statements between two processors a communication link has to be created. In this case the interconnection network will consist of a single unidirectional link,* $proc_1 \rightarrow proc_2$*, used to send the values from array* `b`*. The system can start working on the next iteration before the previous one(s) are finished. At any time (excluding the startup) the computer system will be processing different iterations in a pipelined fashion.*

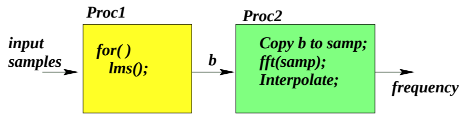

Figure 2: 2-processor architecture implementing the main loop from example 1.

The mapping presented in this example corresponds to the marked point in figure 3 and results in an embedded system presented in figure 4. This system features 2 MOVE processors. Both processors are optimized to the parts of the application which have been mapped onto them, and therefore differ in number of used functional units and busses. They are equipped with small instruction caches and local memories. Also a fast bidirectional interprocessor communication link has been included; each *COMM* unit contains a FIFO supporting asynchronous communication.

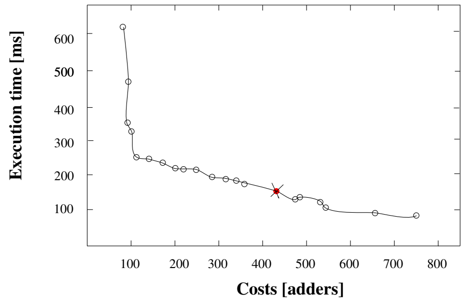

Figure 3: A design trade-offs curve; each point represents a cost efficient multi-processor configuration.

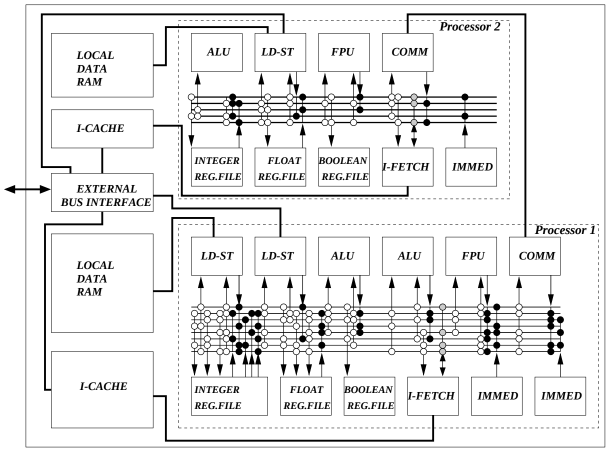

Figure 4: The instant frequency tracking multi-processor.

Probably the most interesting subproblem within the multi-processor design space exploration is that of finding the optimal parallelization of an application. Therefore in

Symbol	Description
N	Number of processors
Σ	Set of parallelization methods used
Θ	Set of parallelized program parts (i.e. loops)
Ξ	Mapping of code & data partitions to processors
Φ	Mapping of data transfers to inter-processor communication hardware
Ψ	Hardware configuration of ASIPs

Table 1: Variables in the design space of the heterogeneous multi-processors.

the following section we will discuss different parallelization methods.

3. Parallelization methods

Basically, we can distinguish two *parallelization modes*. They can be defined as follows:

Definition 1 *In* **operation-parallel** *mode different operations of a single threaded program are executed in parallel. The* **data-parallel** *mode involves applying one or more operations to many data items in parallel.*

Figure 5 demonstrates these modes when applied to a single FOR loop with 9 iterations containing three operations A, B, C. Notation A_i denotes an instance of the operation A executed in iteration i. Parallelization can be obtained by either partitioning the execution of this loop in a horizontal or vertical direction. In the first case iterations $i = 1..3$ would be executed on processor 1, iterations $i = 4..6$ on the second one, etc. (*data-parallel mode*). Alternatively, the vertical partitioning may result in the *operation-parallel mode* in which operations A, B, C may be executed in parallel (A on the first processor, B on the second one, etc.).

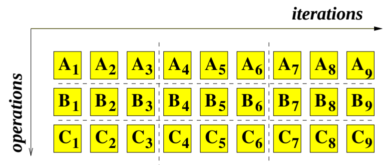

Figure 5: Operation-parallel and data-parallel parallelization modes - vertical and horizontal partitioning.

In case of loops, *operation-parallel* mode leads to *functional pipelining* [12] (as in example 2), while *data-parallel* mode to *index set partitioning* [17]. The following two examples further demonstrate these concepts. Let Λ be a computer system consisting of N processing elements (or shortly processors).

Example 3 (Data-parallel mode)
Index set partitioning *(IDXP) maps a loop in a program, onto Λ in such a way that each processor is executing a subset I_k of all iterations ($\sum_{I_k} = I$, where I is the set of all iterations). Figure 6 shows an example IDXP parallel version of the second loop from the body of the* **lms** *routine (figure 7) as executed on each processor. In the parallel code,* `p` *is the processor id (0..3) and a shared memory interface will be used.*

```
p=my_processor_id;
for (i=32*p; i<32*(p+1); i++ ) {
   b[i]=b[i]+mu*px[i];
}
```

Figure 6: Index set partitioning - parallel code for each of the processors.

Both parallelization modes can of course be applied simultaneously to different loops in the program. Even hierarchical parallelizations are possible as shown in example 4:

```
static float px[128];

for(i=0;i<=128;i++)
  y=y+b[i]*px[i];
Calculate mu;
for(i=0;i<128;i++)
  b[i]=b[i]+mu*px[i];
for(i=128;i>=1;i--)
  px[i]=px[i-1];
```

Figure 7: The simplified source code of the **lms** procedure.

Example 4 (Hierarchical parallelization)
Figure 7 shows the simplified source code of the **lms** *procedure. Starting from the FP parallelization of the main loop, as in example 2, we could obtain larger speedup by applying IDXP to the three loops in* **lms**. *These loops could be mapped onto 2 processors (in the way demonstrated in example 3), resulting in the architecture shown in figure 8. In this configuration the parallel versions of the loops in* **lms** *are executed in parallel by the first and the second processor.*

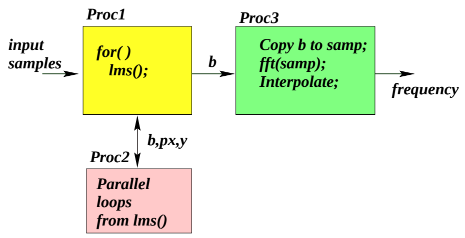

Figure 8: A 3-processor architecture for the embedded application from example 1.

Note that not only the loops of a program suit well for parallelization. Also straight-line code sections may often be parallelized (coarse-[1] and fine-grain).

Often direct parallelization is not advantageous. Code transformations [17] are necessary to enable efficient parallelization. For example, to allow a better *index set partitioning* of a loop nest, a combination of loop interchange and tilling can be used. If we take these code transformations into account then parallelization methods can be defined as follows:

Definition 2 *A* **parallelization method** *is an element of the following set:*

$$\{P\} = \{T\}^* \otimes \{M\},$$

where T is the set of the code transformations and M the set of the parallelization modes.

4. Design Space Exploration

Our design space exploration trajectory takes as input the system specification in ANSI C, accompanied by several parameters, as for example the maximal number of processors available and a set of parallelization methods to be attempted (as defined in section 3). Pushing the design in a certain direction is possible as well, for example by specifying that some program fragments have to be parallelized in a certain way. Such extra directives prune the search space. In addition, our parallelization algorithm requires providing profiling information. It can be obtained by code profiling on an oversized ASIP architecture and with software compilation options for maximal ILP exploitation (the performance metrics obtained in this way represent optimistic bounds).

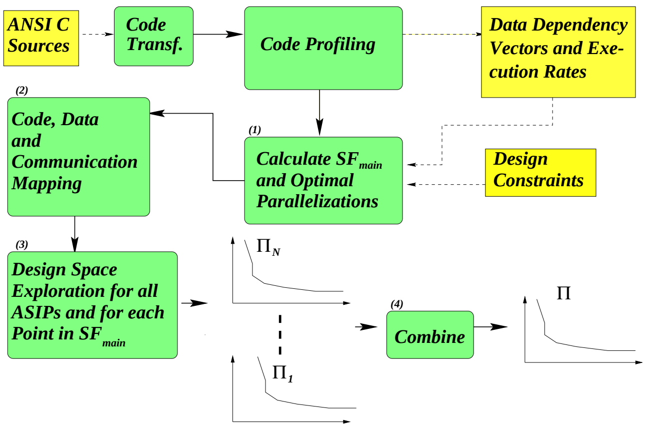

Figure 9: The design space exploration trajectory.

To explain our approach, we define the following:

[1] Assuming that the straight-line code may contain procedure calls.

Definition 3 *A context graph $C(V, E)$ of a program is a directed graph with tree kinds of vertices (procedure, loop and block). Block vertices represent code segments between loops and procedure calls. There is an edge between two vertices if they are nested within each other, or if one calls another. The* root *vertex of $C(V, E)$ is the procedure $main()$.*

Definition 4 *A* speedup function *$SF_v(N)$ of vertex v gives the local speedup at this vertex as a function of the number of processors used at v (example speedup function can be seen at the top of figure 11).*

4.1. System design space exploration trajectory

The system design exploration trajectory is shown in figure 9. It has been optimized to avoid repeatedly performing the most time consuming tasks in every step. Therefore the general code transformations[2], data dependency analysis and profiling are done at the very beginning. For the same reasons the final parallelization and code generation is performed at the end only.

The design trajectory involves the following four major steps:

1. Calculate the speedup function for the *main* procedure: for each number of available processors N, find a set of applied parallelization method(s) Σ and a set of parallelized program fragments Θ which optimize the overall speedup. As side effect the speedup functions for all other vertices in $C(V, E)$ are calculated.

2. Code partitions for each point in the speedup function, which are the result of the parallelization, are mapped onto available processors, data transfers onto inter-processor communication links (Ξ, Φ). The code partitions are mapped onto the processors using the assumption that the serial parts of the program are always executed on processor 1. At parallel regions boundaries the code partitions are assigned to available processors.

3. For each point in the speedup function $SF_{main}(N)$, a design trade-offs curve Π_N between the cost and the performance is calculated (by exploring the design spaces of all processors).

4. The Π_N curves are combined to form a single design trade-offs curve Π (recall figure 3).

4.2. Parallelization

The central place in the whole trajectory occupies a procedure which calculates the speedup function $SF_{main}(N)$.

[2] A transformation is general if it can be applied to most loops, independent of their structure.

In this step the speedup functions for all vertices in the program are calculated recursively, by visiting vertices in the *context tree* of the program in *depth first search* order. After the speedup functions of the successors vertices are calculated the parallelizations at the vertex v are attempted. Three different situations are possible (see figure 10.a,b,c; notation $Part_v(R,N)$ denotes a solution obtained for N processors and parallelization p with R partitions at the vertex v, $R \leq N$):

1. *Vertex-only parallelization*: only the vertex v itself is parallelized (figure 10.a); i.e. $R = N$.

2. *Successors-only parallelization*: only the successor vertices of v are parallelized (figure 10.b); i.e. $R = 1$.

3. *Hierarchical parallelization*: the vertex v itself is parallelized, in combination with successor vertices (figure 10.c); i.e. $1 < R < N$.

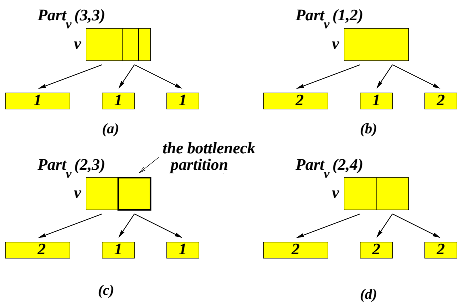

Figure 10: Parallelizations at v

Figure 11 presents the space of the possible parallelizations, together with a speedup function for node v. The hierarchical parallelizations (the third situation) lie within the shaded triangle, the parallelizations of the first group on the diagonal, while the *successor-only parallelizations* on the N-axis. The dashed lines point to the points in the speedup function which are defined by taking maximum over the points lying on these lines in the parallelizations space.

At v we consider only hierarchical parallelizations in the *operation-parallel mode*. This is not an essential limitation because of two reasons:

- A parallelization of a loop nest with n loops is not considered a hierarchical parallelization and is attempted.

- It is expected that in most cases hierarchical *data-parallel mode* parallelizations do not deliver better results than the single level ones. By parallelizing outermost loops we obtain coarser-grain parallelism, with lower communication and synchronization overhead[3].

[3]An exception might be the case where the outermost loop contains fewer iterations than there are processors. However this is very uncommon in our design domain.

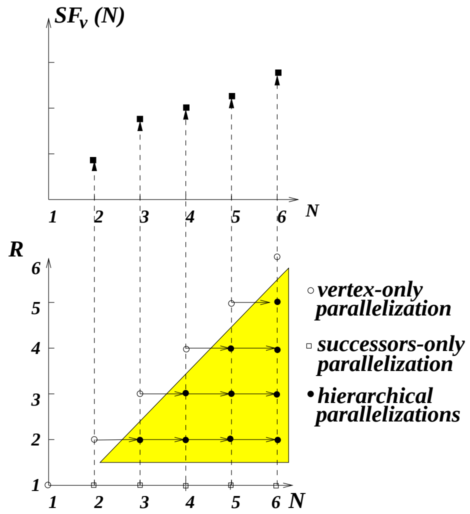

Figure 11: The space of possible parallelizations at v.

Following the division from the beginning of this section, all parallelization try-outs at vertex v can be divided into three groups. First the *vertex-only parallelizations* are considered. The appropriate points from the speedup functions of the successor vertices are selected and the speedup S calculated. Subsequently parallelizations $p \in P$ (recall def. 2) on $N = 2..N_{max}$ processors are tried. Within that group, first parallelization of the vertex v itself is attempted (a point on the diagonal in figure 11). Then, if the parallelization method is *operation-parallel* the hierarchical ones are tried. To avoid high computational complexity, we follow the methodology demonstrated in example 5:

Example 5 (Hierarchical parallelizations)
Suppose that we are at a point to find optimal parallelization of a vertex v onto 4 processors, with two partitions at v (i.e. $Part_v(2,4)$). We can start with the previously obtained parallelization $Part_v(2,3)$ (see figure 10.c). Since the parallelization method p is operation-parallel *both partitions at v will be executed in parallel. One of these partitions will probably have larger latency (in the figure it appeared to be the right one). Only by accelerating this "bottleneck" partition further speedup can be obtained. Therefore we allocate one extra processor to one of the successor vertices of that partition (figure 10.d).*

Following this methodology, in each step the optimal previous solution $Part_v(R, N-1)$ is used as the starting point (following arrows in figure 11). We identify the bottleneck partition at v and attempt using an extra processor on its successor vertices to further speedup v. The overall computational complexity of the parallelization algorithm is $O(N_{max}^2 \cdot |V| \cdot |P|)$. The interested reader is referred to [13] for the detailed description of the algorithm.

5. Experimental results

To verify the effectiveness of our algorithm in parallelizing typical embedded applications we ran it on the set of ASP (Audio Signal Processing) programs taken from [6] and on the MPEG-2 video encoder program from SSG [15]. Table 2 presents the characteristics of all tested programs.

5.1. Measurement method

In the experiments we used a combination of tools belonging to the MOVE automatic processor generation framework [5], and the SUIF parallelizing compiler [1]. For functional pipelining a set of new tools operating on SUIF (Stanford University Intermediate Format) has been implemented [12].

First, we compiled all programs with the **gcc-move** compiler, then scheduled it with the MOVE scheduler [9]. The generated code was simulated using the move simulator to obtain detailed profiling information. The detailed data dependency vectors were generated using the combined static & dynamic methodology described in [14]. Communication overhead was estimated assuming the availability of fast bidirectional interprocessor links.

5.2. Results and evaluation

Initially we checked how much speedup could be reached by exploiting ILP on a uniprocessor system. The speedups obtained by scheduling the code (including software pipelining) on an oversized processor are shown in figure 13. As can be seen in figure 13 the ILP speedups on a uniprocessor system are typical for this class of applications. It is a well known fact that the DSP programs contain many loops with a relatively easy to exploit inter-iteration parallelism. Still these values are much lower than the values predicted by the trace analysis. This forms the strong motivation for further coarse-grain parallelism exploration.

Therefore starting from these fastest ILP optimized solutions, we concentrated further on the coarse grain parallelism exploitation. Our parallelization programs were ran several times for each of the benchmarks from table 2. We tried different sets of parallelization methods and configuration options. The value of $N_{max} = 4$ was used. The first experiment involved calculation of the speedup functions using a combination of two parallelization modes (*data- and operation-parallel*) and assuming no communication overhead model. The resulting speedup functions are presented in figure 14. As reference the fastest ILP optimized uniprocessor solutions were used.

What is not visible in this plot is that actually each point represents an architecture (including all information listed in table 1). For instance, the point for $N = 4$ of the frequency tracking (`instf` in table 2) benchmark from example 1 contains a link to the architecture shown in figure 12. This architecture is the result of applying of *functional pipelining* to the main loop of the program, and of *index set partitioning* of loops in **lms**, **fft** and **main** procedures.

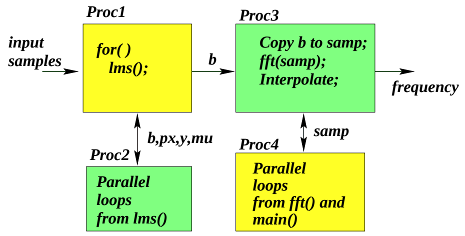

Figure 12: The 4-processor architecture for the frequency tracking embedded system.

In two following experiments we calculated the speedup functions using parallelization methods in one of the two modes only. The obtained speedups for $N = 4$ are presented in figure 15 (DP means *data-parallel* mode and OP *operation-parallel* mode). Based on these speedups we can make conclusions about the importance of different *parallelization modes* exploited by our system. In most cases the *data-parallel* mode plays a larger role in the overall acceleration. But not always. For the g722 and mulaw benchmarks, for instance, the *operation-parallel* mode dominates. Almost all test cases clearly show that the combination of both parallelization modes scores best. Also the average speedups presented in figure 16 show the same tendency. The average speedup for $OP\&DP$ and $N = 4$ amounts to 2.95. This confirms our assumption that only a system being able to exploit all different kinds of fine- and coarse-grain parallelism can deliver consistently good solutions.

Finally, consider the issue of the communication overhead. In all experiments until now we assumed zero communication and synchronization overhead. This is obviously not realistic. Still the obtained numbers are quite valuable, because the results show the upper bounds on the obtainable speedup which can be reached in quite many cases. Consider the following example:

Example 6 *Recall the simplified ANSI C source code of the* **lms** *procedure (figure 7). If we parallelize these three loops in the* data-parallel *mode, and in addition partition the* `b` *and* `px` *array equally between the processors then the total communication overhead will be neglectable. It will include only sending of the final value of the reduction variable* `y` *in the first loop, distribution of* `mu` *before the second loop and sending of single values between the boundaries of the partitions mapped to processors in the third loop. All other read and write accesses will be local on each processor.*

This example demonstrates the importance of the proper data partitioning. The problem is not easy, especially when considered in combination with the choice of parallelization methods and loops to be parallelized. Traditional approaches [4] where the data partitioning was done before the parallelization may lead to suboptimal solutions. In the future we plan to extend our parallelization algorithm to simultaneously perform data partitioning. For the time being we

Benchmark	Description	Operations	Lines	Input data
arfreq	Autoregressive freq. estim.	15M	367	audio sample
g722	Adaptive differential PCM	13M	891	audio sample
instf	Frequency tracking	3M	436	audio sample
interp3	Sample rate conversion	4M	504	audio sample
mulaw	Speech compression	440K	207	audio sample
music	Music synthesis	50M	321	audio sample
radproc	Doppler radar processing	32M	387	audio sample
rfast	Fast convolution using FFT	3M	559	audio sample
rtpse	Spectrum analysis	2M	388	audio sample
mpeg2enc	MPEG-2 encoder	5G	23K	50 video frames

Table 2: Benchmark characteristics. Source size (lines) is given excluding header files and library code.

implemented a simple communication overhead estimator, based on the model in which all arrays are mapped to a single processor and send via a fast bidirectional interprocessor links to the other processors. The estimator assumes that the data transfers (up to a certain quantity) can be scheduled simultaneously with the computations within the loops. The results for $N = 4$ are shown in figure 17. As can be seen in most cases the use of this pessimistic communication model leads only to small performance degradation. Because this model is quite pessimistic in comparison with the one assuming distributed data (recall example 6) the obtained values can be considered lower bounds on the values of the speedup.

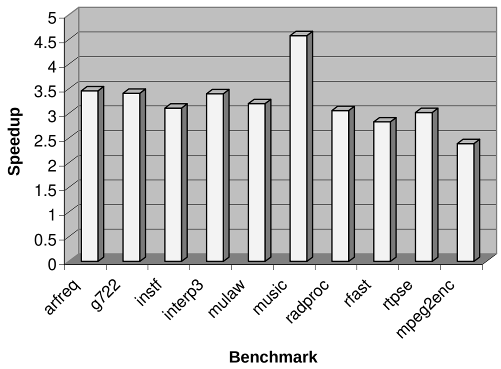

Figure 13: The ILP speedups on uniprocessor system for all benchmarks.

6. Conclusions

In this paper we presented results obtained with our system for semi-automatic mapping of the embedded application onto a cost-efficient heterogeneous multi-processors. Uniqueness of the system lies in a combination of the state of the art automatic ASIP synthesis software with a coarse- and fine-grain parallelism exploitation methodology. Its applicability was demonstrated and evaluated on a number of DSP benchmarks. The obtained parallelization results are consistently good in all test cases. The average speedup for 4 processors is 2.95 relative to an oversized uniprocessor.

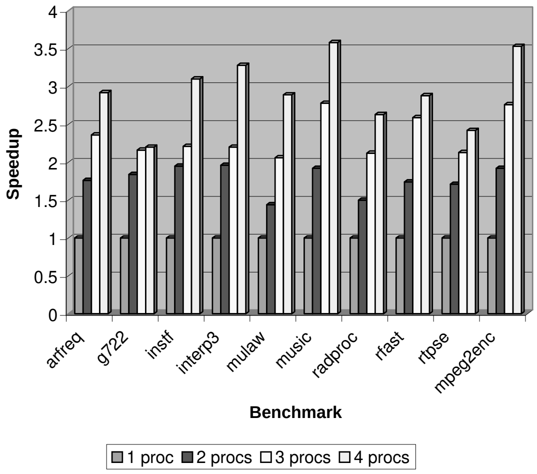

Figure 14: The speedup functions for all benchmarks (relative to an oversized uniprocessor).

The combined fine- and coarse-grain speedups are presented in figure 18 (average speedup on 4 processors equals 9.6). Performed experiments confirm our assumption that only a system being able to exploit all different kinds of fine- and coarse-grain parallelism can deliver consistently good solutions. Our tools bring us very close to the place where the automatic generation of multi-processor embedded systems from the ANSI C specification becomes possible.

References

[1] S. P. Amarasinghe, J. M. Anderson, C. S. Wilson, S.-W. Liao, B. R. Murphy, R. S. French, M. S. Lam, and M. W. Hall. Multiprocessors From a Software Perspective. *IEEE micro*, pages 52–61, June 1996.

[2] J. Axelsson. Architecture synthesis and partitioning of real-time systems: A comparison of three heuristic search. In *International Workshop on Hardware-Software Codesign*, Braunschweig, Germany, March 1997.

[3] A. Bender. Design of an optimal loosely coupled hetergeneous multiprocessor system. In *Proceedings of The European Design & Test Conference*, 1996.

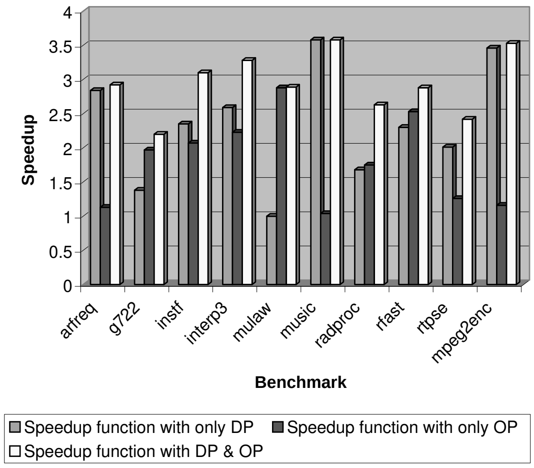

Figure 15: Speedup for $N = 4$ and different parallelization methods (relative to an oversized uniprocessor).

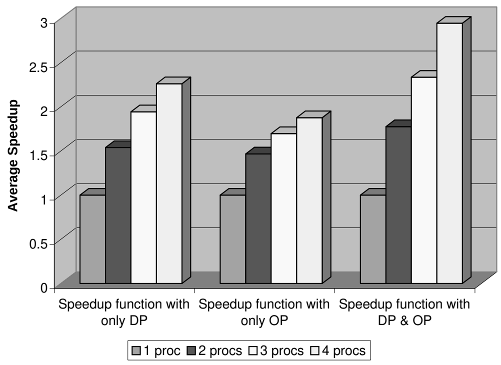

Figure 16: Average speedups for different parallelization methods (relative to an oversized uniprocessor).

[4] R. Bixby, K. Kennedy, and U. Kremer. Automatic data layout using 0-1 integer programming. In *Proceedings of PACT'94*, Montreal, August 1994.

[5] H. Corporaal. *Microprocessor Architectures; from VLIW to TTA*. John Wiley, 1997. ISBN 0-471-97157-X.

[6] P. Embree. *C Language Algorithms for Real-Time DSP*. Prentice-Hall, 1995.

[7] D. D. Gajski, F. Vahid, and S. Narayan. *Specification and design of embedded systems*. PTR Prentice-Hall, Englewood Cliffs, 1994.

[8] R. Gupta. *Co-synthesis of hardware and software for digital embedded systems*. Kluwer Academic, Boston, 1995.

[9] J. Hoogerbrugge. *Code generation for Transport Triggered Architectures*. PhD thesis, Delft Univ. of Technology, Feb. 1996.

[10] I. Karkowski. *Performance Driven Synthesis of Digital Systems*. PhD thesis, Delft University of Technology, December 1995.

[11] I. Karkowski. Computer aided embedded systems design. In *Proceedings of the third annual conf. of ASCI*, Heijen, The Netherlands, 2-4 June 1997.

[12] I. Karkowski and H. Corporaal. FP-Map - An Approach to the Functional Pipelining of Embedded Programs. In *Proceedings of 4th International Conference on High Performance Computing*, Bangalore, India, 18-21 December 1997.

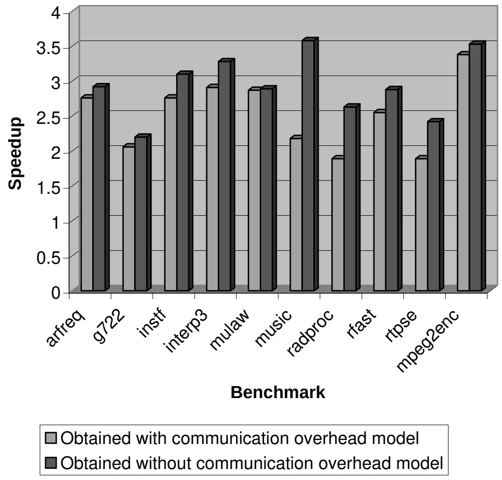

Figure 17: Speedup for $N = 4$ with and without communication overhead model (relative to an oversized uniprocessor).

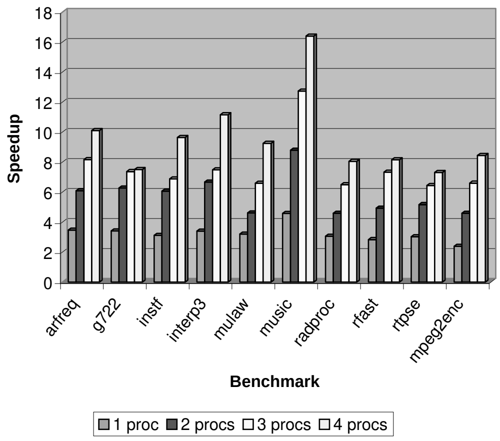

Figure 18: The total speedups for all benchmarks (relative to the single issue uniprocessor).

[13] I. Karkowski and H. Corporaal. Design space exploration algorithm for heterogeneous multi-processor embedded system design. In *35th Design Automation Conference Proceedings*, San Francisco, CA, USA, June 1998.

[14] I. Karkowski and H. Corporaal. Overcoming the limitations of the traditional loop parallelization. *Future Generation Computer Systems*, 13(4-5):407–416, 1998.

[15] MPEG Software Simulation Group, http://www.mpeg.org/index.html/MSSG/#source. *MPEG-2 Video Codec*, 1996.

[16] J. Teich, T. Blickle, and L. Thiele. An evolutionary approach to system-level synthesis. In *International Workshop on Hardware-Software Codesign*, Braunschweig, Germany, March 1997.

[17] M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Publishing Company, 1996.