Design Space Exploration Algorithm For Heterogeneous Multi-processor Embedded System Design

Ireneusz Karkowski and **Henk Corporaal**
Delft University of Technology
Information Technology and Systems
Mekelweg 4, 2628 CD Delft, The Netherlands

Abstract

Single-chip multi-processor embedded system becomes nowadays a feasible and very interesting option. What is needed however is an environment that supports the designer in transforming an algorithmic specification into a suitable parallel implementation. In this paper we present and demonstrate an important component of such an environment - an efficient design space exploration algorithm. The algorithm can be used to semi-automatically find the best parallelization of a given embedded application. It employs functional pipelining [13] and data set partitioning [16] simultaneously with source-to-source program transformations to obtain the most advantageous hierarchical parallelizations.

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 [4]), while the fine-grain (instruction level) parallelism [9] is exploited locally by each of the processors.

In the past several environments for the embedded system design have been realized (a lot of references can be found in [7, 8, 12]). Also a number of papers specifically about multi-processor system design have been published [3, 2, 15]. None of them however addresses the problem of the automatic extraction of the parallelism from the system specification.

In this paper we propose a new approach to mapping of an embedded application written in ANSI C onto a cost-efficient heterogeneous multi-processor. Its uniqueness 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. Section 2 states the problem. Section 3 is devoted to the introduction of different parallelization methods. The system design space exploitation algorithm is presented in section 4. The performance of the algorithm is demonstrated on the frequency tracking system in section 5. Section 6 concludes the paper.

2 Problem statement

Multi-processor system design involves finding a mapping Γ of a program graph $G_P(V, E)$ onto an architecture template graph $G_A(P, C)$, 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, E the set of data and control dependencies between the statements, P the set of processing elements 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.

DAC 98, June 15-19, 1998, San Francisco, CA USA
ISBN 1-58113-049-x/98/06…$5.00

Our goal is to find a design trade-offs curve Π, connecting a set of Pareto points in the 2-dimensional cost-performance space [11] for a given application (see example in figure 10; as the cost usually the chip area, power dissipation or number of packaging pins is used). Having it, we can select a cost-efficient and feasible (satisfying the constraints) solution.

Symbol	Description
N	Number of processors
Σ	Set of parallelization methods used
Θ	Set of parallelized program parts (i.e. loops)
Ξ	Mapping of 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.

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 1 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 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.). Note that not

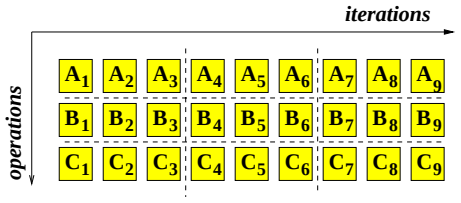

Figure 1: Operation-parallel and data-parallel parallelization modes - vertical and horizontal partitioning.

only the loops of a program suit well for parallelization. Also straight-line code sections can often be parallelized (coarse- and fine-grain).

Often direct parallelization is not advantageous. A series of code transformations [16] 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 algorithm 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 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, the 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).

To explain the algorithm, we define the following:

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 $main()$. With every vertex a coefficient $l(v)$ is associated. It represents the total number of clock cycles (latency) required to execute its code (including successor vertices). Also every edge carries one coefficient - $f(e)$, which takes a value different than 1 in case it targets a procedure vertex for a procedure called from more than one place in the program. In such case this value represents a fraction of the vertex latency which is due to that calling site.*

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 functions can be seen in figure 8).*

4.1 System design space exploration algorithm

The algorithm for system design exploration is shown in figure 2. It has been optimized to avoid repeatedly performing the most time consuming tasks in every step. Therefore the

general code transformations[1] and data dependency analysis are done at the very beginning. For the same reasons the final parallelization and code generation is performed at the end only.

The algorithm performs the following four major steps:

1. After some initialization code, 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 barriers (moments when parallel execution is started or terminated) 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, see figure 9).

4. The Π_N curves are combined to form a single design trade-offs curve Π (see figure 10).

```
Apply a set of general code transformations;
Do data dependency analysis;
Build the context graph C(V,E);
for ( all v ∈ V and N = 1 ... N_max ) SF_v(N) = 1;
SF_main=SPEEDUP(main);
Map code partitions to processors, data transfers
  to communication links for points in SF_main;
Do design space exploration of the processors
     for points in SF_main;
Combine results into a design trade-offs curve Π;
```

Figure 2: System design space exploration algorithm

```
SpeedupFunction SPEEDUP(v){
   for ( all { w | w = succ(v) ∧ l(w)/l(main) ≥ α} )
      SF_w=SPEEDUP(w);
   PARALLELIZE(v, SF_v);
   Return SF_v;
}
```

Figure 3: The SPEEDUP function

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

The function called SPEEDUP is presented in figure 3. As can be seen, the speedup functions for all vertices in the program are calculated recursively. Only vertices with $p(v) = \frac{l(v)}{l(main)} \geq \alpha$ are considered. The value of the coefficient α should exclude vertices which form a marginal part of the total computation time. Accelerating vertices with low $p(v)$ is very unlikely to deliver any significant speedup even if many processors are used.

4.2 Parallelization

After the speedup functions of the successors are calculated (first loop inside the SPEEDUP procedure), the parallelizations at the vertex v are attempted. Three different situations are possible (see figure 4; 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 (top-left); i.e. $R = N$.

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

3. *Hierarchical parallelization*: the vertex v itself is parallelized, in combination with successor vertices (bottom-left and bottom-right); i.e. $1 < R < N$.

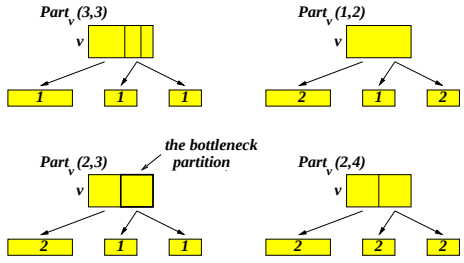

Figure 4: Parallelizations at v

Figure 5 presents the space of the possible parallelizations, together with a speedup function for 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*[2]. This is not an essential limitation; it is expected that in most cases hierarchical *data-parallel mode* parallelizations do not deliver better results than the single level ones.

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

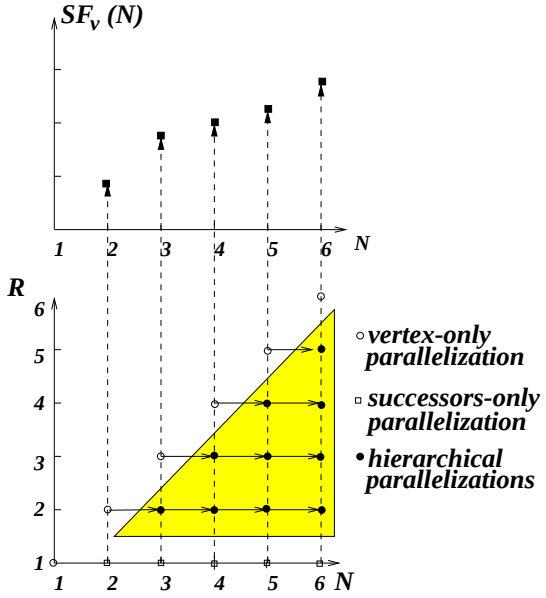

Figure 5: The space of possible parallelizations at v.

The number of processors used at vertex v, $N_{proc}(v)$ can be calculated using the following formulas:

$$N_{proc}(v) = \sum_{b \in B(v)} N_{proc}(b) \quad (1)$$

$$N_{proc}(b) = \max_{\{w | w=succ(v) \wedge w \subset b\}} N_{proc}(w) \quad (2)$$

where $B(v)$ is the set of partitions at the vertex v and $N_{proc}(b)$ the number of processors used inside successor nodes of the partition b.

To obtain exact results for the hierarchical parallelizations a substantial number of alternatives (combinations of the points in the speedup functions of the successor vertices) would have to be considered. It can be shown that the number of alternatives K can be calculated using the following formula:

$$K = \sum_{N=2}^{N_{max}} [2 + \sum_{R=2}^{N-1} (N - R + 1)^z] \quad (3)$$

where R is the number of partitions at v and z the number of successor vertices of v. For example if $z = 6$ and $N_{max} = 6$ then, using this formula, we obtain $K = 26270$ combinations. In practice this number will be slightly smaller since some alternatives cannot result in a legal parallelization, but still we would have to run our parallelization algorithms for the majority of them. This can very easily result in unacceptably long run times.

The following example shows a more practical method of obtaining a legal parallelization:

Example 1 *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 the bottom-left part of figure 4). 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 (bottom-right part of figure 4).*

To avoid the high computational complexity we decided to use an approximation algorithm to calculate the best parallelizations at v (the PARALLELIZE procedure in figure 6). In its body, the *apply(v,p,R)* procedure applies a parallelization method p at v with R partitions at v and returns the obtained parallelization X. The obtained speedup depends on the parallelization method used and is calculated using the *sp(p,X)* procedure.

First, appropriate points from the speedup functions of the successor vertices are selected and the speedup S calculated (first FOR loop). Subsequently parallelizations $p \in P$ (recall def. 2) on $N = 2..N_{max}$ processors are tried. In the first inner loop, a parallelization of the vertex v itself is attempted. Then, if the parallelization method is *operation-parallel* the hierarchical ones are tried. We follow the methodology from the example 1. The optimal previous solution $Part_v(R, N-1)$ is used as the starting point (following arrows in figure 5). We identify the bottleneck partition at v and attempt using an extra processor on its successor vertices to further speedup v.

```
PARALLELIZE(v, SF_v) {
  /***** successors-only  parallelization *****/
  for (N = 2; N ≤ N_max; N = N + 1) {
    S = l(v)/ Σ_{w|w=succ(v)} l(w)/SF_w(N)
    SF_v(N)=max (SF_v(N), S);
  }
  for ( all p ∈ P ) {
    for (R = 2; R ≤ N_max; R = R + 1) {
  /***** vertex-only parallelization **********/
      for ( all {w | w = succ(v)} ) Select SF_w(1);
      X=apply(v, p, R);
      SF_v(R)=max (SF_v(R), sp(p, X));
      if ( p is operation-parallel ) {
  /***** hierarchical parallelization  ********/
        for (N = R + 1; N ≤ N_max; N = N + 1) {
          Identify bottleneck part. b at v in X;
          for ( all {w | w = succ(v) ∧ w ⊂ b} )
             Select next point in SF_w;
          X=apply(v, p, R);
          SF_v(N)=max (SF_v(N), sp(p, X));
  }}}}
}
```

Figure 6: Parallelization of the vertex v

4.3 Computational complexity

The maximal number of the parallelization try-outs necessary per vertex and per parallelization method is a small constant, which depends only on the number of points in the speedup function (equal to the number of points in the area $R \geq 2$ in

figure 5):

$$Y = \sum_{n=2}^{N_{max}} (n-1) = \frac{N_{max} \cdot (N_{max} - 1)}{2} \qquad (4)$$

For example for $N_{max} = 6$ we obtain $Y = 15$ tries, which is much less than the number of combinations possible (recall eq. 3). This together with the fact that each vertex is visited only once has a positive effect on the total computational complexity of the algorithm. This complexity is $O(N_{max}^2 \cdot |V| \cdot |P|)$.

5 Case study

In this section we present an example application of our algorithm - the frequency tracking embedded system from [6]. The system specification contains about 2k lines of ANSI C source code. In its main loop, the program reads a stream of samples (complex numbers) 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.

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 [13].

First, we compiled the frequency tracking program with the **gcc-move** compiler, then scheduled it with the MOVE scheduler [9]. The options for maximal ILP exploitation (including software pipelining) and oversized architecture (large number of move busses and FUs) were used. The generated code was simulated using the move simulator to obtain detailed profiling information. A number of general code transformations has been applied to this code. For example the *lms()* and *fft()* functions were inlined inside the FOR loop bodies *F20* and *F15*. The context graph including computation distribution information is presented in figure 7. For the sake of readability, only the loop and procedure vertices are shown. Vertices which are marked with '*' are suitable for *data-parallel mode*, while the ones with '#' for *operation-parallel mode* parallelization. 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 only.

Subsequently the SPEEDUP procedure was called to generate the speedup functions. The following parameters were used: $N_{max} = 4,\ \alpha = 0.01$. Several vertices with $p(v) < \alpha$ were skipped (F57, F91, F121, F72). Total of 297 parallelizations were attempted. The generated speedup functions for the most important vertices in the context graph are presented in figure 8. As can be seen from the speedup function for the whole program (SF_{main}), an overall speedup varying from

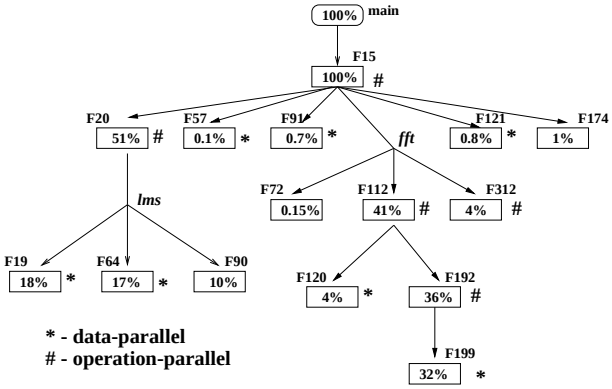

Figure 7: Context graph for the frequency tracking system

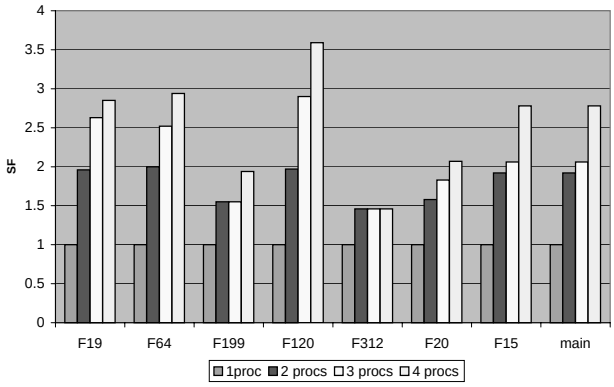

Figure 8: Speedup functions for selected vertices from figure 7

1.92 to 2.78 could be obtained. The point for $N_{proc} = 2$ involves *operation-parallel* mode parallelization of the F15 loop, while the speedup of 2.78 on 4 processors can be obtained by applying in addition parallelizations to the loops F19 and F64 in *lms*, and to the loops F120, F199 and F312 in *fft*.

Next the design trade-offs curves Π_N for $N = 1..4$ were generated using the *Explore* tool of the MOVE framework. Figure 9 presents these curves. To combine them we have to select Pareto points from each curve. In our case this results in the design trade-offs curve presented in figure 10. Note that points for implementations with larger number of processors lie to the right of the implementations with smaller N.

After obtaining the combined speedup function one of the points had to be selected. The specified timing constraint was 150 ms (required speedup of 1.48). The point for the configuration with only 2 processors turned out to be sufficient (marked in figure 10). Note, that none of the single processor solutions, even with many FUs, meets the timing constraints. The obtained multi-processor is presented in figure 11. Besides standard components, processor 1 included 2 ALUs, 1 FPU, 2 Load-Store units and 8 move busses. The

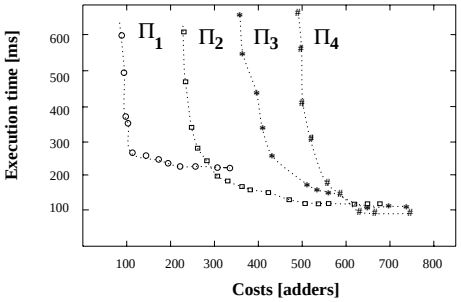

Figure 9: The design trade-offs curves for different numbers of processors used.

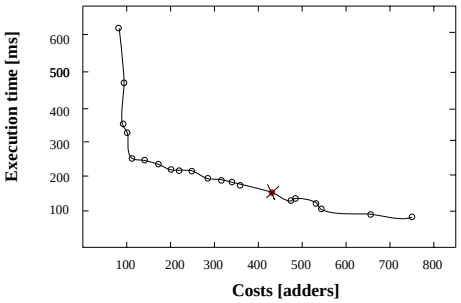

Figure 10: The combined design trade-offs curve.

second processor turned-out to be smaller - included only 1 ALU, 1 FPU, 1 Load-Store unit and 5 move busses. Both processors were 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. Note also that the multi-ported (integer) register file can easily be split into four dual ported register files (see [10]).

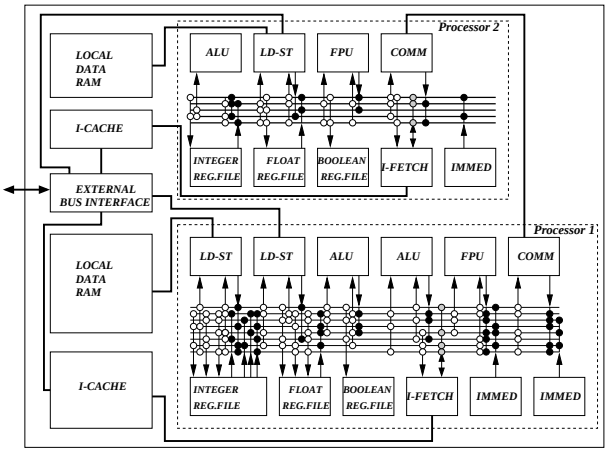

Figure 11: The instant frequency tracking multi-processor.

6 Conclusions

In this paper we proposed a new design space exploration algorithm for semi-automatic mapping of the embedded application onto a cost-efficient heterogeneous multi-processors. Its uniqueness lies in a combination of the state of the art automatic ASIP synthesis software with a coarse- and fine-grain parallelism exploitation methodology. The computational complexity of the algorithm is linear in the number of loops in the program and in the number of applied parallelization methods. Its applicability was demonstrated on a case study of the frequency tracking system.

The presented approach can be easily extended to handle real-time reactive embedded systems with many subtasks. We can consider each subtask separately when calculating the speedup functions. After that, perform the mapping of the task partitions onto the processors and do the processors design space exploration. Once the trade-offs functions are calculated we should check them for possible subtask constraints violations and then combine them and select points which satisfy the global constraints.

References

[1] Saman P. Amarasinghe, Jennifer M. Anderson, Christopher S. Wilson, Shin-Wei Liao, Brian R. Murphy, Robert S. French, Monica S. Lam, and Mary 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.

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

[5] Henk Corporaal and Hans Mulder. MOVE: A framework for high-performance processor design. In *Supercomputing-91*, pages 692–701, Albuquerque, November 1991.

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

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

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

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

[10] Johan Janssen and Henk Corporaal. Partitioned Register Files for TTAs. In *MICRO-28*, Michigan, November 1995.

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

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

[13] Ireneusz Karkowski and Henk 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.

[14] Ireneusz Karkowski and Henk Corporaal. Overcoming the limitations of the traditional loop parallelization. In *Proceedings of the HPCN'97*, April 1997.

[15] 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.

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