FP-Map - An Approach to the Functional Pipelining of Embedded Programs

Ireneusz Karkowski and **Henk Corporaal**
Delft University of Technology

Abstract

Practice shows that increasing the amount of instruction level parallelism offered by an architecture (like adding instruction slots to VLIW instructions) does not necessary lead to significant performance gains. Instead, high hardware costs and inefficient use of this hardware may occur. Mapping embedded applications onto multiprocessor systems forms a very interesting extension to ILP. In this paper we propose a functional pipelining approach to the mapping of embedded programs written in ANSI C onto a pipeline of application specific processors. Our novel functional pipelining algorithm has low computational complexity and was especially developed to form the parallelization engine of a (semi)-automatic system for multiprocessor embedded system design. The paper explains the proposed algorithm and demonstrates its applicability.

Keywords: heterogeneous multi-processors, parallelization techniques, high performance embedded system design

1 Introduction

Previous research has proven the existence of the large potential of fine-grain [22] and coarse-grain [14] parallelism in sequential programs. Unfortunately this potential is not always easy to exploit. In practice, when exploiting exclusively the instruction level parallelism (ILP), only average speedups in the range of 2-5 are observed [13]. This is far less than the parallelism upper bounds predicted by trace analysis ($\approx$60) [22]. We are confronted with a fundamental question about the limitations of the ILP exploitation methods. The obstacles seem to lie both in the area of software parallelization techniques, as in the architecture of the nowadays ILP processors (i.e. superscalar, VLIW). We will analyze them in section 2.

Due to high complexities of the nowadays embedded systems the design automation frameworks for them become a "must". What is needed is an interactive environment that supports the designer in transforming an algorithmic specification into a suitable implementation. The environment should allow fast experimentation with different architectural options and release the designer from the burden of many time consuming, but often simple, synthesis tasks. Several such embedded system design environments have been created in the past [10, 12, 4]. They differ significantly in many aspects. For example in the choice of the application specification language, target architecture selection and/or hardware/software partitioning algorithms. Clearly however the general design flow inside these environments exhibits several similarities. The interested reader is referred to [15] for the description of the most important aspects of the embedded system design.

One of the most challenging parts in this design process is the architecture selection step. What we need is a set of algorithms which can be used for the coarse-grain partitioning of embedded applications onto different multiprocessors architectures. In the past, several parallelizing compilers for symmetric shared memory multi-processors were implemented [3]. They partitioned the iteration space of the loops in a sequential program among available processors. Code transformations were usually necessary to ensure correct parallel execution of those partitions. Unfortunately, in the context of the embedded systems, often the symmetric shared memory architecture is not cost-efficient, because it results in large hardware duplication and requires a complex memory system. Also, exploiting only data parallelism in loops may not be sufficient to justify the use of the multi-processor framework. Some other, coarse-grain parallelism exploitation methods, oriented towards simpler multi-processor architectures, are necessary.

This paper is devoted to one such parallelization technique - the so called "functional pipelining". The technique could be shortly defined as a mapping of a selected loop nest in a sequential program onto a pipeline of processors, in such way that a pipelined style of execution is obtained (analogous to the way automobile assembly line works). We use an efficient algorithm for automatic "functional pipelining" of a selected loop nest in a program written in ANSI C onto a pipeline of the application specific processors [16]. While applicable to any loop nest, the technique was especially designed to map loop nests laying very high in the call hierarchy of a program and containing all kinds of control statements. We use it as a component in our multi-processor embedded system design framework.

The remainder of the paper is organized as follows. Section 2 analyzes the obstacles hampering further exploitation of ILP on uni-processor systems, and forms a motivation for our approach. In section 3 we provide necessary definitions and formally state the problem. Section 4 is devoted to the short presentation of our mapping algorithm called

FP-Map. We have tested it on many embedded programs. Its performance is demonstrated on an example loop in section 5. In section 6 we discuss how to extend the scope of the application of the algorithm. Section 7 concludes the paper.

2 Limitations of the ILP

In the past, numerous ILP exploitation methods were proposed and used. Among them, the techniques for the ILP in loops received most attention. The software pipelining [20, 1] is the most popular one in this class. It has been successfully applied during code generation for the VLIW processors. Unfortunately the method has its limitations. Namely, few software pipelining algorithms are able to effectively schedule loops which lie high in the call tree of the program and/or contain control constructs. This significantly limits their application scope. Handling of conditional statements received most attention. In the past, several solutions have been proposed. *Hierarchical reduction* [18] hierarchically reduces conditional statements to single nodes having resource requirements being the union of both branches and the latency of the longer branch. If a much longer branch is much less frequently executed, a very inefficient schedule results. The *All Paths Pipelining* [21] extracts different iteration paths in a loop body, applies software pipelining to each path in isolation and inserts code that is executed from one schedule to another when a subsequent iteration executes a different path. While in many cases quite successful, the algorithm exhibits for some loops a substantial code expansion. Another software pipelining algorithm, called *Enhanced Pipeline Scheduling* [8, 9], uses extremely aggressive speculative scheduling combined with register renaming to pull up as many future computations as possible, including loop-back branches. Disadvantage of this approach is its inability to deal with complex resource patterns [2]. Inefficiencies in the kernels generation have been reported as well. In addition, on architectures where the FU pipelining is visible at the architectural level (as is the case with our MOVE ASIP processors [7, 13]) the implementation of the *Enhanced Pipeline Scheduling* is tedious.

The conditionals are not the only high level language constructs hampering the effective software pipelining. Loops and function calls present in the loop's body form even a greater challenge. While loop unrolling and function inlining can be used in some cases, no good universal solution exists. Application of all these methods may result in a substantial code expansion, which is usually undesired in the "embedded" context.

Another more fundamental problem arises. To reach the parallelism upper bounds as predicted by trace analysis, all possible control paths through the program should be scheduled separately in parallel (emulating multiple flows of control [19]). Clearly, this leads to disastrous code size increase, and becomes even infeasible in the presence of the loops with compile time unpredictable loop counts. This problem is serious.

Also on the hardware side several problems tempering effective exploitation of the ILP exist. The most important one is that of the complexity of the internal interconnection network of the known ILP processors. It increases rapidly with the number of available FUs. For example on VLIW processors this network complexity is quadratic in the number of FUs ([6], chapter 4). The run-time instruction scheduling hardware of the superscalar processors reaches complexities difficult to handle even faster. This results in large area consumption and puts limitation on the achievable cycle times. The MOVE (Transport Triggered Architecture) processors, remove most of the limitations of the traditional ILP processors. Still however, by increasing their number of available FUs, the load on the buses obviously increases and therefore the transport delays. This load can be reduced, but exclusively by some very sophisticated architectural measures (for example internal clustering of FUs, [6], section 7.4.2).

Clearly, at a certain moment the limit on the ILP exploitation is reached. Further progress is only possible by a combined fine- and coarse-grain parallelism exploitation. The advent of sub-micron processing, allowing integration of several millions of transistors on a single carrier, has brought new opportunities in the embedded system design. The embedded multi-processors become nowadays a feasible and very attractive option. Especially if the system consists of heterogeneous processors, each having the functionality optimized for the part of the computation that it has to perform. This way both fine- and coarse-grain parallelism can be efficiently exploited.

3 Preliminaries

Our problem could be formally stated as that of 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. The problem of finding a good mapping for arbitrary G_P and G_A is clearly too complex. The problem which we address in this paper is simpler, however, still far from being trivial. We start namely with a subgraph of G_P containing a set of vertices from one of the loop nests in the program. Also an architecture template graph is given (namely a linear pipeline of ASIPs). Our goal is to find a mapping, which optimizes the performance of the system.

Let us start with few definitions. Usually there exists a number of data dependencies between statements in a loop body. The calculation of these dependencies in embedded programs written in ANSI C has been discussed in [17]. The data dependencies can be divided into two categories:

Definition 1 *The* intra iteration *data dependency is a dependency between the executions of two statements belonging to the same loop iteration. Conversely, the* inter iteration *dependency exists between statements executed in different iterations. A* backward inter iteration *dependency is an* inter iteration *dependency going from a statement b to a,* *with the property that there exists a path $a \rightarrow b$ composed of the* intra iteration *data dependencies.*

Definition 2 *Let Λ be a computer system consisting of N_{proc} processing elements (or shortly processors), connected with a number of unidirectional and asynchronous communication links.* Functional pipelining *is a mapping that maps a graph G_L containing vertices from the body of a loop nest in a program (a subgraph of G_P), onto Λ in such a way that the resulting architecture is processing several iterations of the loop in a pipelined fashion.*

```
forall (samples) {
  a=do_fft(sample);
  b=freq_mult(a,c[sample_id-2]);
  c[sample_id]=bandpass(b);
  }
```

Figure 1: Example loop nest

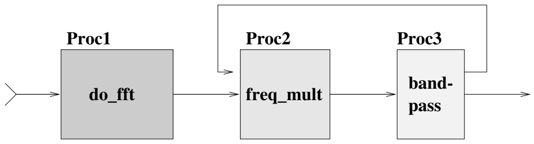

Figure 2: The 3 processor architecture implementing the code from figure 1

As an example, consider the sequential program in figure 1. This loop can be mapped onto a 3-processor computer system, in which the *do_fft* function is mapped onto the first processor in the chain, the *freq_mult* onto the second one, and *bandpass* operation onto the third one (figure 2). Clearly when we partition data dependent statements in G_L between two processors in Λ, a communication link has to be created. In this case the interconnection network will consist of three unidirectional links, $proc_1 \rightarrow proc_2$, $proc_2 \rightarrow proc_3$ and $proc_3 \rightarrow proc_2$. 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 3 different samples in a pipelined fashion. The existing *backward inter iteration* dependency going from the third to the second statement requires some attention. Thanks to the fact that the iteration distance of this dependence is equal 2, no pipeline stalls (assuming a balanced processor load) are necessary - when the second processor starts processing iteration i, the third processor will be already done with iteration $i-2$ and starting iteration $i-1$. As a result the value of $c[sample\_id-2]$ can be retrieved via the backward communication link between both processors. Note that this would not be possible without a stall if the iteration distance of this dependency was 1.

Clearly, by successfully applying *functional pipelining* to parts of an embedded program substantial speedup can be obtained. In the next section we will show how this mapping can be found, and how to estimate the speedup.

4 The methodology

As stated in definition 2 our goal is to map a body of a loop nest onto a computer system Λ consisting of N_{proc} processing elements. This mapping consists of the following three steps:

1. Reduce the loop dependency graph $G_L(V, E)$ to $G_{LR}(V, E)$.
2. Map G_{LR} to Λ.
3. Estimate the speedup.

In the following subsections its steps will be discussed in detail.

4.1 Reduce the loop dependency graph

The reduced dependency graph $G_{LR}(V, E)$ is a directed graph, where V is the set of vertices and E the set of edges. The name "reduced" comes from the fact that it models groups of operations from the G_L by single vertices, and that it excludes some "relatively unimportant" operations (as will be described in section 4.2). With every vertex and edge a number of coefficients is associated. In case of vertices we have a 5-tuple (l, ρ, c, g, b) and in case of edges a single value $\vec{p}$. $l(v)$ is the time necessary to perform the operation(s) modeled by the vertex (in clock cycles), $\rho(v)$ is the probability that a given operation(s) will be executed during a single iteration, $c(v)$ the id of the conditional statement to which the vertex belongs or $\perp$ if not conditional, $g(v)$ is the value of the conditional statement test expression, which selects the branch where v belongs, $b(v)$ is equal $\top$ if the vertex lies on the longest branch of the conditional, $\perp$ otherwise and $\vec{p}(e)$ the iteration distance vector.
Vertices in G_{LR} are created according to the following *vertex creation rules*:

- All loops (including all statements within it) and call statements are mapped into single vertices. The latency l of these vertices is set to the maximum time that an execution of this loop or call takes.
- For every conditional statement we create a *test* vertex with latency l being the time necessary to calculate the test expression. Then we include all vertices on the branches (recursively applying the *vertex creation rules*). Also the values of c, g, b are appropriately set for all the vertices.
- All other groups of statements located between the above mentioned constructs are mapped into single vertices with the latency calculated as the latency of the most critical path in the instruction level data dependency graph.

To each node the execution probability $\rho(v)$ is attached as measured during profiling. For edges we only have a single value $\vec{p}$ which is the dependence distance. An edge between vertex u and v denotes that the operation corresponding to v in iteration $\vec{i}$ should be executed at least $l(u)$ clock cycles after the operation corresponding to u in iteration $\vec{i} - \vec{p}$. There is also an edge between two vertices if there exists a control dependency between those vertices. We assume that every vertex $v \in V$ represents an atomic computational entity to be allocated on a processor, and that the instructions inside it will be scheduled separately (exploiting the ILP). Therefore by the longest branch we mean the branch B with maximal sum of vertex latencies $\sum_{v \in B} l(v)$.

4.2 Additional graph reduction

As we wrote in the previous section the G_{LR} is a "reduced" graph in the sense that multiple instructions from the G_L are mapped onto its vertices. Additional reduction of the graph can be obtained by skipping vertices which are "relatively unimportant" - have low execution probability $\rho(v)$ and/or small latency $l(v)$. Considering these vertices would lead to situations where unnecessarily long schedules are generated in cases when we have a much longer branch which is almost never executed (well known problem experienced in software pipelining). We approach it by optimizing the "most common case". In its "stable" state the pipeline will only execute the "most common case" statements. When one of the other branches is triggered, the whole pipeline will possibly have to be stalled. To obtain such an effect, only vertices with high enough execution probability $\rho(v)$ will be considered. Further reduction can be obtained by merging some "relatively short" vertices with their predecessors. This leads to the following definition of the *the MCC set* (most common case set):

$$V_{MCC} \equiv \{v \in V |\ \rho(v) \geq \alpha \wedge \frac{|V| \cdot \rho(v) \cdot l(v)}{\sum_{u \in V} \rho(u) \cdot l(u)} \geq \beta\} \quad (1)$$

where α and β are a user given cut values. In practice the value of α should be low enough to avoid excluding any essential computation. Also the value of β should be carefully selected to avoid precision loss.

4.3 Perform the mapping

The mapping problem is NP-complete [16], and therefore we decided to use an efficient heuristic. The algorithm could be shortly described as a combination of the *modulo scheduling* algorithm used for *software pipelining* and the FFD (first fit decreasing) algorithm used for the *bin packing problem* ([11], p.226). We could call our method a "processor based" scheduling (as opposite to the "cycle" or "operation" based schedulers). In place of the resource constraints only a single constraint on the maximum load on a processor is considered[1]. The algorithm consists of the following steps:

1. Calculate the set of *strongly connected components* (SCC) of G_{LR}. This information is later used to determine if a given SCC may be partitioned among processors. For example, if it has a cycle ϕ with $p_m = \sum_{e \in \phi} p(e)$ then all its vertices can be distributed among at most p_m processors.
2. Calculate upper and lower bounds (L_{min}, L_{max}) on the initiation interval II of the pipeline. This can be done in very much the same way as in *modulo-scheduling*. The only extensions include the usage of the "longest branch" information present in G_{LR}. For example the upper bound can be calculated as follows:

$$L_{max} = \sum_{v \in V} (c(v) = \perp ? l(v) : (b(v) = \top ? l(v) : 0)). \quad (2)$$

   where expression $(expr\ ?\ value_1 : value_2)$ should be understood as follows: if $expr$ is true then the value of the whole expression is $value_1$, otherwise $value_2$.
3. Perform a search over this range. In every step a "packing" procedure attempts to schedule and pack (in FFD style) all vertices of the G_{LR} inside N_{proc} processors (one at the time), each with capacity L $(L_{min} \leq L \leq L_{max})$. In every scheduling step the candidate vertex is selected from a priority list of "ready" vertices. The priority is based on the longest path from each vertex to one of the sink vertices in G_{LR}. The algorithm selects a vertex belonging to a cycle in G_{LR}, only if its assignment to the current processor will not cause a violation of the data dependencies. If successful it returns a partitioning P and sets value of II to L. Otherwise, the obtained partitioning (with remaining nodes allocated to the last processor) and $L' > L$ is returned. This way the upper bound value L_{max} is automatically tightened, speeding up the search process.
4. Use the "packing" with minimum value of L as the solution to the partitioning problem.

Unfortunately space limitations do not allow us to present here the full details of our mapping algorithm. Interested reader is referred to [16].

4.4 Computational complexity

It is easily determined that the computational complexity of this algorithm is low; and therefore it is suitable for the application within the architecture selection step, where a huge number of alternatives has to be investigated. The set of *strongly connected components* of $G_{LR}(V, E)$ can be calculated in $\mathcal{O}(|V| + |E|)$ time using the algorithm from [5], page 489. The longest paths from every vertex to all sinks can be calculated by the modification of the Bellman-Ford algorithm [5], page 532, which runs in $\mathcal{O}(|V||E|)$ time. The values of L_{min} and L_{max} can be calculated in $\mathcal{O}(|V|)$ time. The number of steps of the search

[1] Other processor resources, like number of functional units do not have to be checked; a processor scheduling node v with latency $l(v)$ cycles can always be produced.

does not directly depend on the problem size, and appears to be low, due to the instant tightening of the upper bound value by the "packing" procedure. The complexity of this procedure depends on the implementation of the *Ready* priority list. If the "heap" implementation is used ([5], page 151), any operation on the list will take $\mathcal{O}(\lg |V|)$ time. Since subsequently $|V|$ vertices are scheduled, this results in the total $\mathcal{O}(|V| \lg |V|)$ complexity for the "packing" procedure. Since $\lg |V| < |E|$, the overall complexity becomes $\mathcal{O}(|V||E|)$.

4.5 Speedup estimation

The estimated latency of the pipelined version of the loop can be calculated using the following equation:

$$L_{fp} = L \cdot (I_{tot} + N_{proc} - 1) + L_{stall} \quad (3)$$

where L is the *initiation interval* of the pipeline, I_{tot} total number of iterations in the loop and L_{stall} the upper bound on the total stall time of the pipeline. The pipeline may stall when some vertices $v \notin V_{MCC}$ are executed (recall section 4.2). The upper bound on total *expected* stall time is:

$$L_{stall} = \sum_{v \notin V_{MCC}} l(v) \cdot \rho(v) \cdot I_{tot}. \quad (4)$$

The estimated speedup which can be obtained by the *functional pipelining* of a given loop nest is then given by:

$$\mathit{SPEEDUP}_{fp} = \frac{I_{tot} \cdot L_{max}}{L_{fp}}, \quad (5)$$

where L_{max} is given by the equation 2.

For various reasons, $\mathit{SPEEDUP}_{fp}$ represents only an estimate:

1. Obviously by the heuristic nature of the mapping algorithm.
2. Not every execution of a vertex $v \notin V_{MCC}$ will cause a pipeline stall. If this happens on a processor which is less loaded then the maximally loaded one, a stall may not be necessary.
3. Actual time which a partition will take to execute may be different due to the local ILP exploitation. Once the real code for every partition will be generated it will be optimized by a code scheduler. Nothing would stop it also from local software pipelining. We try to take these effects into account by using the profiling information for already scheduled and heavily optimized executable. After the mapping the location of some operations may however completely change, leading to difficult to predict effects. In addition in the model we did not yet account for the communication overhead.

5 Mapping example

In this section we will present an example application of our algorithm. Consider the loop nest in figure 3 and $N_{proc} = 3$. The G_{LR} is presented in figure 4. There are two SCCs

```
for(i=0;i<10;i++)
  for(j=0;j<20;j++){
    int a,b,c,d,e,f,g,h,k;
    a=A[i][j];
    b=bkx(B[i][j]);
    U[i][j]=a;
    if(a>2){
      c=C[i][j-2]%3+rrl(0x0234);
      C[i][j]=2*c;
      d=C[i][j]-2;
    } else {
      E[i][j]=E[i][j]+0x01;
      d=E[i][j]%5;
    }
    D[i][j]=tew(d,4);
    if(a>b){
      e=R[i][j]-A[i][j];
      f=sqrt(e)+wt(A[i][j]);
    } else {
      g=L[i][j-1]-A[i][j];
      k=h*1.28+wt(g);
      I[i][j]=sqrt(k);
      L[i][j]=sin(k);
      f=sqrt(L[i][j])*0.2;
    }
    s=f/L[i][j];
  }
```

Figure 3: An example C source code. For readability reasons all the variables were renamed and privatized to avoid any *anti* and *output* dependencies

with $|s| > 1$ in the graph, $SCC_1 = \{c, C\}$ with $\sum p = 2$ and $SCC_2 = \{g, k, L\}$ with $\sum p = 1$. Also two conditional statements are present with *test* vertices $t1$ and $t2$. During profiling we observed the branch probabilities close to 0.5 for both conditionals. Therefore no vertices were excluded from the graph (i.e. $V = V_{MCC}$). The longest branches of the conditionals are: $br_1 = \{c, C\}$ with total latency of 57 and $br_2 = \{g, k, I, L, f\}$ with total latency of 68. The range of the search interval was calculated to be:

$$L_{min} = 57, \quad L_{max} = 170.$$

First run of the "packing" procedure for the value of $L = 57$ resulted in a failure, but it produced a new upper bound value $L_{max} = 82$. After a number of steps of the search we arrived at the final value of $L = 65$. The thick lines in figure 4 denote the obtained partitioning boundaries.

Now the speedup can be calculated using the formula 5:

$$\mathit{SPEEDUP}_{fp} = \frac{200 * 170}{65(200 + 3 - 1)} = 2.59$$

6 Extensions

As could be seen in the previous section the *functional pipelining* technique is expected to be successful in parallelization of the large loop nests, lying high in the call hierarchy of the program. Unfortunately, in the majority of the embedded programs written in ANSI C such loop nests do not occur often. Some special measures have to be taken to extend the application scope of the algorithm. In the discus-

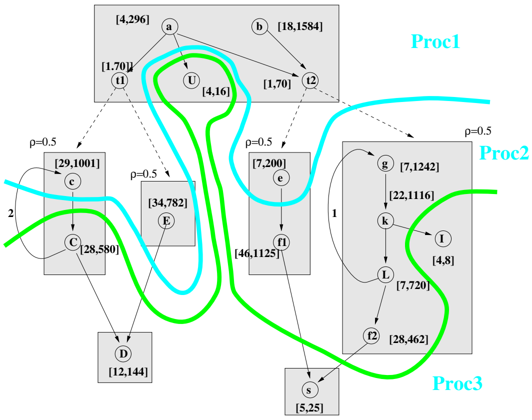

Figure 4: The G_{LR} for the example source code from figure 3. Only some coefficients are presented: $[l(v), priority(v)]$ for vertices, and p's for only backward edges. Dashed arrows represent control flow.

sion below we will use the concepts of the code transformations as described in [23]. The following optimizations can increase the application scope of the **FP-Map** algorithm:

- A loop nest may be not tightly nested (i.e. a loop is not the only statement in the body of another loop). The problem can be handled by applying the code transformation called *code sinking*.
- Often nested loops are distributed among calling procedures. *Call inlining* allows the increase of the depth of the loop nest.
- Several loop nests with similar iteration space may follow each other in the program. By performing *normalization* and *loop fusion* (also *loop skewing* may be necessary) they can be merged together to form a single large loop.

7 Conclusions

In this paper we proposed an approach to the *functional pipelining* of the embedded programs written in ANSI C. The problem of increasing the parallelism exploitation ratio in the sequential programs has been discussed. We concluded that this goal can only the reached by a combined fine- and coarse-grain parallelism exploitation. A multiprocessor embedded system framework has been recognized as a very interesting and cost-effective extension to a single processor embedded system. We proposed an algorithm for mapping the embedded programs written in ANSI C onto a pipeline of ASIP processors. The algorithm has computational complexity of $\mathcal{O}(|V||E|)$ and was especially developed for the parallelization of the loop nests laying very high in the call hierarchy of a program and containing all kinds of control statements.

References

[1] Alexander Aiken and Alexandru Nicolau. Optimal Loop Parallelization. In *Proceedings of the SIGPLAN'88 conference on Programming Language Design and Implementation*, pages 308–317, Atlanta, Georgia, June 1988.

[2] V.H. Allan, R.B. Jones, R.M. Lee, and S.J. Allan. Software pipelining. *ACM Computing Surveys*, 27(3):367–432, September 1995.

[3] 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.

[4] P. Chou, R.B. Ortega, and G. Borriello. The chinook hardware/software co-synthesis system. In *International Symposium on System Synthesis*, Cannes, France, September 1995.

[5] T.H. Corman, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. McGraw-Hill, MIT-Press, 1990.

[6] Henk Corporaal. *Transport Triggered Architectures; Design and Evaluation*. PhD thesis, Delft Univ. of Technology, September 1995. ISBN 90-9008662-5.

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

[8] Kemal Ebcioğlu. A Compilation Technique for Software Pipelining of Loops with Conditional Jumps. In *Proceedings of the 20th Annual Workshop on Microprogramming*, December 1987.

[9] Kemal Ebcioğlu and Toshio Nakatani. A New Compilation Technique for Parallelizing Loops with Unpredictable Branches on a VLIW Architecture. In *Proceedings of the Second Workshop on Programming Languages and Compilers for Parallel Computing*, University of Illinois at Urbana-Champaign, 1989.

[10] R. Ernst and J. Henkel. Hardware-software codesign of embedded controllers based on hardware extraction. In *International Workshop on Hardware-Software Codesign*, Boston, September 1992.

[11] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of the NP-Completeness.* Freeman, San Francisco,CA, 1979.

[12] R.K. Gupta, C.N. Coelho, and G. De Micheli. Synthesis and simulation of digital systems containing interacting hardware and software components. In *Proc. of DAC'92*, pages 225–230, 1992.

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

[14] Jeroen Hordijk and Henk Corporaal. The Impact of Data Communication and Control Synchronization on Coarse-Grain Task Parallelism. In *Second Annual Conf. of ASCI*, Lommel, Belgium, June 1996.

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

[16] Ireneusz Karkowski and Henk Corporaal. Fp-map - an approach to the functional pipelining of embedded programs. Technical Report 1-68340-44/1997/06, Delft University of Technology, Delft, The Netherlands, April 1997.

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

[18] Monica Lam. *A Systolic Array Optimizing Compiler*. The Kluwer International Series in Engineering and Computer Science. Kluwer Academic Publishers, Norwell, Massachusetts, 1989.

[19] Monica S. Lam and Robert P. Wilson. Limits of control flow on parallelism. In *ISCA-92*, pages 46–57, Australia, May 1992.

[20] B. R. Rau and C. D. Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. In *Proceedings of the 14th Annual Workshop on Microprogramming*, pages 183–198, October 1981.

[21] Mark G. Stoodley and Corinna G. Lee. Software pipelining loops with conditional branches. In *IEEE micro*, pages 262–273, 1996.

[22] David W. Wall. Limits of Instruction-Level Parallelism. In *Proceedings of ASPLOS-IV*, pages 176–188, Santa Clara, California, April 1991. ACM.

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