

An efficient microcode-compiler for custom DSP-processors*

Gert Goossens

Jan Rabaey[†]

Joos Vandewalle

Hugo De Man

IMEC Laboratory, B-3030 Leuven, Belgium

ABSTRACT - In this paper, a microcode compiler for custom DSP-processors is presented. This tool is part of the CATHEDRAL II silicon compiler. Two optimization problems in the microcode compilation process are highlighted: microprogram scheduling and memory allocation. Algorithms to solve them, partly based on heuristics, are presented. Our compiler successfully handles repetitive programs, and is able to decide on hardware binding. In most practical examples, optimal solutions are found. Whenever possible, indications of the complexity are given.

1 Introduction

The CATHEDRAL II silicon compiler aims at the automatic synthesis of synchronous multi-processor ASICs for DSP-applications [3] [4]. It proposes a full separation between architecture synthesis and layout generation (meet-in-the-middle philosophy). The architecture synthesis tools transform an applicative behavioural representation of a DSP-algorithm, into a structural description of multi-processor data paths [3], and a procedural microcoded controller definition. In a first step, a customized data path, composed of well-characterized parametrizable modules, is generated by the rule-based synthesis program JACK-THE-MAPPER [9]. At the same time, the behavioural representation is translated into an applicative register-transfer (RT) description. In a second step, the microcode compiler ATOMICS¹ transforms the applicative RT-program into a procedural finite-state machine description. The output of this tool is linked with a PLA-based controller compiler, performing state optimization and layout generation, and with an RT-level simulator. In this paper we will describe the microcode compiler ATOMICS, with emphasis on efficient optimization algorithms for microprogram scheduling (section 2) and memory allocation (section 3).

The applicative RT-input language of ATOMICS is characterized by a highly architecture-independent format, and powerful control constructs, e.g. FOR-loops and conditional statements. In many compilers, loop constructs are not supported since they cause certain optimization problems to become hard. In the sequel, RT-programs (not) containing loops will be referred to as (non-) repetitive programs. ATOM-

ICS is based on a parametrizable multiple-branch controller model with horizontal microcode, allowing for simultaneous data path actions and jump address computation [3]. The current version of ATOMICS does not perform any transformations, which create, delete or alter RTs from the input description.

2 Microprogram scheduling

In order to capture the time-concept in a repetitive RT-program, the potential of an RT is defined as the RT's machine-cycle number in an imaginary implementation, in which the body of every FOR-loop is executed once. The goal of microprogram scheduling then is to map the individual RTs to potentials, thereby exploiting the inherent parallelism in the DSP-algorithm in order to minimize the global machine-cycle count. In order to reduce the complexity of the scheduling problem for repetitive RT-programs, loops are scheduled hierarchically, starting at the deepest level of nesting.

2.1 A constrained optimization problem

The search space of the optimization is bounded by three categories of constraints:

1. *Forward data-precedences*, expressing the need for a minimal (integer) delay δ between any pair of RTs A and B , respectively writing and reading a variable to/from a storage element. This can be denoted:

$$p_B(j) \geq p_A(j) + \delta(j) \quad (1)$$

where $p_A(j)$ and $p_B(j)$ represent the potentials of A and B respectively, in constraint j . Usually, δ equals 1 cycle. Dependencies between conditional RTs and RTs producing the according condition variables are also modelled in this way. In this case δ is usually larger than 1, due to the internal controller pipelining.

2. *Resource-allocation constraints* (or *conflicts*), expressing the requirement that certain RTs sharing hardware resources (e.g. data path operators, memory, busses) cannot be scheduled at the same machine cycle. If C and D represent the conflicting RTs in constraint i , the latter can be denoted:

$$p_C(i) \neq p_D(i) \quad (2)$$

3. *Looping data-precedences*, expressing a precedence relation in a FOR-loop, between a writing RT in the current

*Research supported by the ESPRIT 97 program of the EC.

[†]Current affiliation: Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA 94720, USA.

¹"A Tool for Optimized Micro-Instruction Compilation and Scheduling".

loop iteration and a *reading* RT in the next iteration. Looping precedence j' between RTs A and B can be denoted :

$$p_B(j') = p_A(j') + \delta(j') - (\Delta p + 1) \quad (3)$$

where $\Delta p + 1$ is the total number of potentials spanned by the loop. (3) is only relevant when $\delta(j') = 1$.

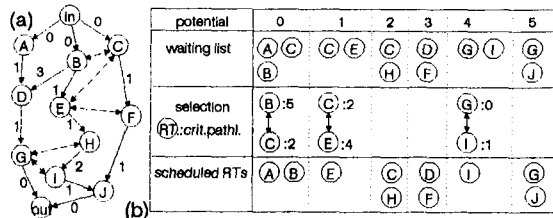


Figure 1: Scheduling of non-repetitive RT-program : (a) data-precedence graph (solid) and resource-allocation graph (dotted). Figures indicate weights of data-precedences; (b) scheduling based on critical path criterion for conflict resolution.

No constraints occur between conditional RTs with disjoint condition fields. Two graph representations will be used (e.g. Fig. 1(a)). In both cases, RTs are modelled as vertices. Forward data-precedences are modelled as arcs in a directed weighted graph, from the writing to the reading RT's vertex, with a weight equal to the delay (*data-precedence graph*). Resource-allocation constraints are represented as arcs in an undirected graph (*resource-allocation graph*).

Scheduling problems are known to be NP-complete. (1), (2) and (3) can e.g. be reformulated as an *integer linear program* (ILP) [3]. Efficient *heuristic scheduling algorithms* from the theory of *project management* [2] have been applied to the DSP-domain by Zeman [10], for the subclass of *non-repetitive RT-programs*. We will present an iterative scheduling algorithm, based on [10], which allows to schedule *repetitive programs* in an automatic and efficient way. In this paper, we assume that storage elements in the data path may contain multiple storage-fields (e.g. RAM, ROM, local register-files). Although not described, our iterative scheduling technique allows to model other memory elements too, such as pipeline latches.

2.2 Scheduling non-repetitive programs

We will review Zeman's scheduling technique, for *non-repetitive programs*, as an extension of classical levelling of the data-precedence graph. The levels of the vertices, correspond to the potentials assigned to the RTs. A global search method (levelling), used to take into account the forward data-precedences, is combined with local heuristic searches to take into account the NP-hard resource-allocation constraints. All potentials are treated consecutively, starting from 0. Inspection of the data-precedence graph reveals which RTs are ready to be assigned the current potential. These RTs are placed in a waiting list. If resource-allocation conflicts occur, a heuristic selection criterion is used to derive a conflict-free subset from the waiting list. The non-selected RTs are shifted to the waiting list of the next (higher) potential. In *ATOMICS*,

a *critical-path criterion* is used : the scheduling priority of an RT involved in a conflict at the current potential, is measured by the length of the critical path in the data-precedence graph, emerging from the RT. Modifications of the criterion are discussed in [6]. The algorithm is illustrated in Fig. 1(b). Compared to a levelling algorithm, only minor extra CPU-time is spent : the complexity of a critical path analysis is linear in the number of vertices in the subgraph under consideration.

2.3 Scheduling repetitive programs

Below, we will extend Zeman's scheduling algorithm, in order to schedule *repetitive programs*. An *iterative procedure* allows to take into account the additional *looping precedences*. The convergence is guaranteed, and acceptable bounds on the number of iterations can be derived.

In the algorithm for *non-repetitive RT-programs*, the *forward precedences* (1) can be interpreted as a specification of a *lower bound on the potential* p_B of an RT B which is ready to be scheduled. This bound is a function of the potentials $p_A(j)$ of all precedent RTs, i.e. all RTs pointing to B via a forward precedence, and of the associated delays. By definition, B is ready to be scheduled when all of its precedents A have been scheduled; hence at this point the bound can be computed explicitly, and equals the potential of the waiting list in which B will occur for the first time. In a *repetitive program*, an additional bound on p_B is provided by the *looping precedences* (3). In this expression, Δp can be estimated (see below); the potentials $p_A(j')$ are however unknown during the treatment of the theoretically optimal potential of B . Therefore, the bound cannot be computed explicitly. In *ATOMICS*, this problem is solved by calling Zeman's algorithm repeatedly in an iteration process : in order to compute the lower bounds (1) and (3) on p_B during iteration k , the potentials $p_A(j')$ are taken from the previous iteration $k - 1$, whereas the potentials $p_A(j)$ are taken from the current iteration, since they are known in time.

According to experiments, the convergence of the iteration process is critically dependent on the estimate of Δp [6] : a too small choice of Δp results in an empty solution space for the optimization, which is manifested as divergence of the iteration. Divergence is *observable* from certain vertices in the graph [6]. A too large choice of Δp results in convergence to a sub-optimal solution. A reliable strategy to determine the optimal Δp is to call the previous algorithm repeatedly in a *second iteration loop* over Δp . The optimal Δp is determined in a limited number of steps, via a *binary* or *incremental* search between a reliable lower and upper bound on Δp . An upper bound on Δp is e.g. the sum of all delays in the precedence graph; this is then also a (pessimistic) bound on the number of outer-loop iterations.

An example is given in Fig. 2. With our algorithm, the optimal schedule has been found in numerous designs. Applications include a 4-processor pitch-extractor for speech, an adaptive interpolator for error correction in Compact Disc, and an echo canceller for digital telephony. Some results are listed in Table 1.

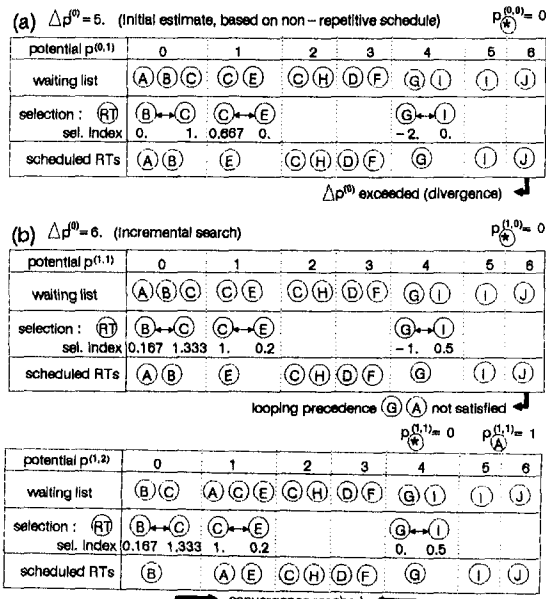


Figure 2: Scheduling of repetitive program, obtained from Fig. 1, by adding looping precedences $G \rightarrow A$ and $out \rightarrow in$, with delays 4 and 1 respectively : (a) 1st outer iteration step, consisting of 1 inner iteration step ; (b) 2nd outer iteration step, consisting of 2 inner iteration steps. (The selection index is a modified critical-path measure.)

Table 1: Design figures for *ATOMICS* scheduler.

RT-system	pitch-extractor					
	pitch computation processor			recognition processor		
# RTs	60			66		
# forward data-precedences	188			162		
# looping data-precedences	51			169		
# resource-alloc. conflicts	46			98		
# potentials	15			16		
cycle count	137			3523		
FOR-loop indices	time	i	j	time	i	y
# inner iterations	1	2	1	1	4	1
CPU-time (VAX8600/VMS)	0.21 s			1.27 s		

2.4 Automated hardware-assignment during scheduling

A designer often might wish to use multiple instances of a certain hardware operator in the data path (e.g. ALUs, multipliers,...). The decision on the number of instances of each operator will be termed the *hardware allocation*. When multiple instances of an operator have been allocated, the next step in the design is to bind each RT to one or more specific instances. This problem will be termed the *hardware assignment*. Below, an optimized hardware-assignment technique will be introduced, which is based on the scheduling algorithm. Full details can be found in [7]. For a given allocation, *ATOMICS* will try to find an assignment which minimizes the machine-cycle count. For this purpose, an RT input-description has to be provided in which all RTs are formally using *differently named operator-instances*, called *formal instances*. *ATOMICS* will then *merge* these formal instances into the actual number of instances, specified in the *allocation*. Automated hardware-assignment is non-trivial with architectures in which single RTs may cover multiple operators.

E.g. in *CATHEDRAL II* data paths, register-files are placed locally at the inputs of operators [3]; therefore the assignment of each RT requires choosing both a *source* operator (for fetching and modifying source data) and a *destination* operator (for storing the result). In general, the assignment of RTs involved in a data-precedence, cannot be performed independently of each other. This fact encumbers the locality of the search process.

In order to combine hardware assignment with scheduling, two types of *resource-allocation conflicts* have to be distinguished : *soft* conflicts, occurring between RTs covering different formal instances of an operator; and (classical) *hard* conflicts, occurring between RTs sharing the same actual instance of an operator. RTs which are involved in a soft conflict, may be scheduled on the same potential, as long as sufficient actual operator-instances are available to execute them in parallel. In this case, the final assignment should obey the condition that the formal instances which caused the soft conflict, may not be merged into one actual instance. Such condition is termed a *merging constraint*.

As in section 2.2, *ATOMICS* will process consecutive potentials, starting from 0. At every potential, with the help of the critical path criterion, a maximal subset of RTs is derived from the waiting list, which contains a) no hard conflicts and b) only soft conflicts that don't introduce an *inconsistent* (unsolvable) *set of merging constraints* for the given allocation. A set of merging constraints can be represented as arcs in an undirected graph, termed *merging graph*, with formal operator-instances as vertices. The problem of checking the consistency of a set of merging constraints at every potential [7] is equivalent to finding an acceptable *vertex colouring* [5] of the merging graph, in which the number of colours corresponds to the the number of allocated operators. Although vertex colouring is NP-complete, the limited size of the merging graph usually allows to apply a simple *branch and bound* algorithm [7]. Every acceptable vertex colouring, of the merging graph obtained after completion of the scheduling operation, provides a valid hardware-assignment. The cheapest colouring in terms of interconnection cost can then be chosen.

3 Memory allocation

When the RT-program has been scheduled, an allocation of variables to storage fields can be carried out, aiming at a minimal number of fields. In *ATOMICS*, this allocation is used to dimension register-files in the data path. The set of potentials during which a variable is to be stored is called the *variable's lifetime* [1]. The lifetime of a variable in a non-repetitive program can be modelled as an *integer interval*

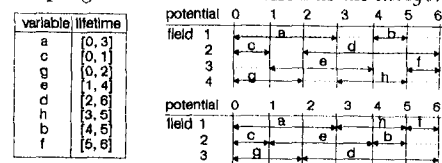


Figure 3: Lifetime intervals of variables in a register-file, an arbitrary (top) and an optimal register-allocation (bottom).

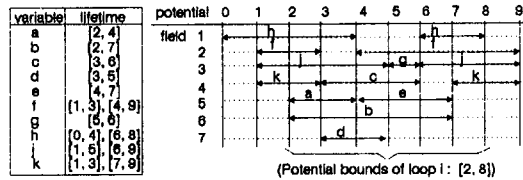


Figure 4: Register-allocation for repetitive program.

(Fig. 3). In the repetitive case however, lifetimes may have to be defined as the *union of non-overlapping intervals* [6] (Fig. 4). One storage field may be allocated to different variables if they have disjoint lifetimes.

The general memory allocation problem is equivalent to *clique partitioning*, which is NP-complete [8]. However, for the class of non-repetitive programs, a *globally optimal* allocation can be found in *linear* time, with the algorithm presented in Table 2. For an optimality proof, we refer to [6]. The analysis is demonstrated in Fig. 3 (bottom).

Table 2: Optimal register-allocation for non-repetitive program.

```

p := minimal potential;
create new storage-field f;
while there exist unassigned variables do
  if there exist unassigned variables  $v_i$ , coming alive at p then
    select arbitrary  $v_i$ ;
    assign  $v_i$  to f;
    p := endvalue of  $v_i$ 's lifetime
  else
    p := p + 1
end if;
if p >= maximal potential then
  p := minimal potential;
  create new storage-field f
end if
end do.

```

This algorithm requires that lifetime intervals be continuous. For the class of repetitive programs, *ATOMICS* uses an efficient heuristic procedure, incorporating the previously described algorithm. This procedure consists of two steps :

- Consider the variables with a lifetime composed of several non-overlapping intervals. A heuristic graph-based algorithm [6] allows to "fill the gaps" in these lifetimes, with smaller continuous lifetime intervals of other variables. Both the "filling" and the "filled" variables are assigned to the same storage-field. In this way, discontinuous lifetimes are formally replaced by single continuous intervals.
- Next, the optimal allocation algorithm for non-repetitive programs is applied.

With this procedure we succeeded in finding the optimum in almost any practical design, in very low CPU-time. An example is given in Fig. 4. As has been observed in [8], direct application of general heuristics for clique partitioning often produces suboptimal solutions to the memory allocation problem. In Fig. 5, a special configuration is shown, for which the results of the basic clique partitioning algorithm presented in [8] are compared with our technique.

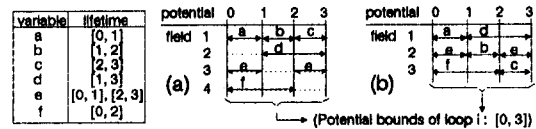


Figure 5: Example of register-allocation, comparing our technique (b) with the approach described in [8] (a).

4 Conclusions

The microcode compiler *ATOMICS* for custom DSP-systems has been presented. The problems of scheduling (with an extension towards automated hardware binding) and memory allocation, have been emphasized. Efficient algorithms have been presented, which are partly based on heuristics, to compute quasi-optimal solutions in very low CPU-time. Loop-constructs in the applicative input description are supported. Current research is aimed at further optimization of repetitive programs, e.g. by allowing overlaps in time between successive loop-iterations. Furthermore, it is investigated how memory optimization can be taken into account during scheduling.

References

- [1] A.V. Aho, J.D. Ullman, "Principles of compiler design", Addison-Wesley, Reading, 1977.
- [2] E.G. Coffman Jr, "Computer and job scheduling theory", J. Wiley and Sons, New York, 1976.
- [3] H. De Man et al., "CATHEDRAL II : a synthesis and module generation system for multiprocessor systems on a chip", *Proc. NATO-ASI on log. synth. and silicon compilation for VLSI-design*, L'Aquila, July 1986.
- [4] H. De Man et al., "CATHEDRAL II : a silicon compiler for digital signal processing", *IEEE Design and Test*, pp. 13-25, Dec. 1986.
- [5] A. Gibbons, "Algorithmic graph theory", pp. 198-201, Cambridge Univ. Press, Cambridge, 1985.
- [6] G. Goossens et al., "An efficient microcode-compiler for custom multiprocessor DSP-systems", *Technical report ESPRIT97/IMEC/3.87/D/d(4)/1*, available from IMEC vzw, Leuven, March 1987.
- [7] G. Goossens, "Techniques for automated hardware assignment", *Technical report ESPRIT97/IMEC/3.87/D/d(4)/1*, available from IMEC vzw, Leuven, Sept. 1987.
- [8] C.J. Tseng, D.P. Siewiorek, "Automated synthesis of data paths in digital systems", *IEEE Trans. CAD*, pp. 379-395, July 1986.
- [9] J. Vanhoof et al., "A knowledge-based CAD system for synthesis of multi-processor digital signal processing chips", *Proc. VLSI'87*, Vancouver, Aug. 1987.
- [10] J. Zeman, G.S. Moschytz, "Systematic design and programming of signal processors, using project management techniques", *IEEE Trans. ASSP*, pp. 1536-1549, Dec. 1983.