Scheduling, Compaction and Binding in a Retargetable Code Generator using Constraint Logic Programming

Ulrich Bieker, Steven Bashford
University of Dortmund, Dept. of Computer Science, 44221 Dortmund, Germany
email: bieker@ls12.informatik.uni-dortmund.de

Abstract - Code generation for embedded programmable processors is becoming increasingly important. Many of these processors have irregular architectures and offer instruction-level parallelism (e.g. DSPs). In order to generate code for a wide range of architectures, a code generator should be retargetable. Most of the previous code generation approaches concentrate on the datapath, not taking the peculiarities of the controller into account. The controller can have strange address generation schemes and imposes restrictions on the amount of parallelism in the datapath. In this paper we propose a new method to model all these restrictions and characteristics of the controller uniformly, in order to perform scheduling, compaction and binding in a retargetable code generator. For this, we exploit the programming paradigm of constraint logic programming (CLP). CLP offers a general and uniform model for various constraints, performs consistency checks, and integrates constraint solving techniques.

1. Introduction and related work

During the recent years, new research activities emerged in the area of code generation for embedded systems [MaGo95]. However, compiler support for embedded processors like DSPs and ASIPs still must be improved to develop high-quality code and to have a retargetable compiler for different architectures. Such a retargetable compiler is very useful, e.g., in the context of hardware-software codesign . A retargetable compiler is target independent and uses a description of the target architecture as input in order to generate code for a certain algorithm. Different approaches can be classified by considering the three main target models which are used.

Most of the code generation approaches use an instruction set model of the processor [Faut95, Hein93, BaHa81]. Instruction set models provide a high level of abstraction but lack in modelling all details of the processors. Contrary approaches use structural models of the processor, e.g., a complete register transfer structure [Nowa87]. Furthermore, mixed models are used, describing the instruction set as well as certain details of the hardware structure [LPKS95, PLMS95]. We propose to start with a structural model of the processor, in order to capture all specific details of the datapath and the controller. An instruction set can be extracted from the structural description [LeMa94, BiNe96].

The task of code generation can be roughly divided in the three following phases: code selection, register allocation, scheduling and compaction. Code selection is a mapping of source language statements to machine operations. Register allocation is a mapping between source program variables and intermediate results to machine registers. Scheduling and compaction determines a total order of the instructions with respect to a) the parallelism of the target machine (e.g. in VLIW architectures) and b) the existing dependencies between the instructions.

Early compaction techniques have been developed for microcoded machines. In [LDSM80, DLSM81], the basic compaction techniques are described, e.g. *linear analysis* [DaTa76], *critical path* [Rama74], *branch and bound* [YST74], and the class of *list scheduling techniques.* The compaction methods rely on heuristics to reduce the search space of possible schedules with the aim of pruning solutions representing no good results. List scheduling is the most popular local technique.

There exists a known bound on the time it takes to execute, the complexity is $O(n^2)$ [LDSM80]. List scheduling produces good results in the presence of adequate heuristics and is easy to implement. A detailed introduction to compaction methods, list scheduling, and a summary of heuristics can be found in [Beat91].

A few classes of instruction scheduling techniques have evolved in the last years (see figure 1). Many principles developed for compaction techniques can be found in these techniques. A disadvantage of local techniques is, that basic blocks generally offer a more limited degree of instruction level parallelism. Therefore architectures that offer a high amount of parallelism (including deeply pipelined machines) are not supported appropriatly. Recent research has succeeded in overcoming the basic block boundaries. Three basic classes of global instruction scheduling techniques have emerged in recent years. *Trace scheduling* [Fish81] is an extended list scheduling technique not restricted to basic block boundaries and employs branching probabilities to select the most likely execution path of a program. The selected path (trace) is then regarded as if it is a basic block. To preserve program semantics when moving instructions across conditional jumps, *compensation code* has to be inserted. *Percolation scheduling* also is a global scheduling technique based on four semantic preserving program tranformation rules performed on the program flow graph (similar to the control flow graph, but with nodes containing concurrent instructrions) [Nico85]. Percolation scheduling reduces the generation of superfluous compensation code; code explotion is a major draw back of trace scheduling. *Region scheduling* is a technique based on the program dependency graph and allows movements of machine instructions over wider program regions than percolation scheduling [GuSo90]. Region scheduling redistributes available parallelism, such that machine resources are fully utilized. This is also achieved by applying semantic preserving transformation rules on the program dependence graph. Region scheduling can also be used for coarse grain scheduling on source code level.

The basic issues of investigations of the techniques and their superimposed and enhanced approaches are:

1. Improvements in effective support of instruction level parallelism. Avoid the generation of superfluous compensation code.
2. Taking resource constraints into account is becoming of much interest. E.g., percolation scheduling assumes unbounded resources. Many efforts are made to integrate the consideration of a restricted number of machine resources [EnNi89, MoEb92, NoNi92] (c.f. Global Resource Constraint Scheduling (GRiP)).
3. Consideration of over-, and under- utilized regions of a program with respect to the underlying target architectures are made (e.g. Region Scheduling, Resource Spacling [GuSo90, BGS95]).
4. Integration of transformations rules that allow the movement of operations accross large program regions (Trailblaizing Percolation Scheduling (TiPS) [NoNi93]).
5. Phase coupling with code selection (Mutation Scheduling).
6. Integration of scheduling accross loops (e.g. TiPS, Software pipelining).

Except for region scheduling, the basic scheduling techniques can only handle acyclic code. In region scheduling, loops are compacted by unrolling its body. This approach can be very inefficient with regards to code space. Therefore, another class of scheduling techniques based on *software pipelining* has been developed. Hereby, a new iteration of the loop is started before the preceeding iteration has completed. Extended versions based on percolation scheduling and region scheduling have been developed to incorporate software pipelining (e.g. *perfect pipelining* [AiNi88], *enhanced pipelining percolation scheduling* [EbNa89], and *enhanced region scheduling* [AJL92]).

The approaches mentioned above do not consider all restrictions of the controller and irregular

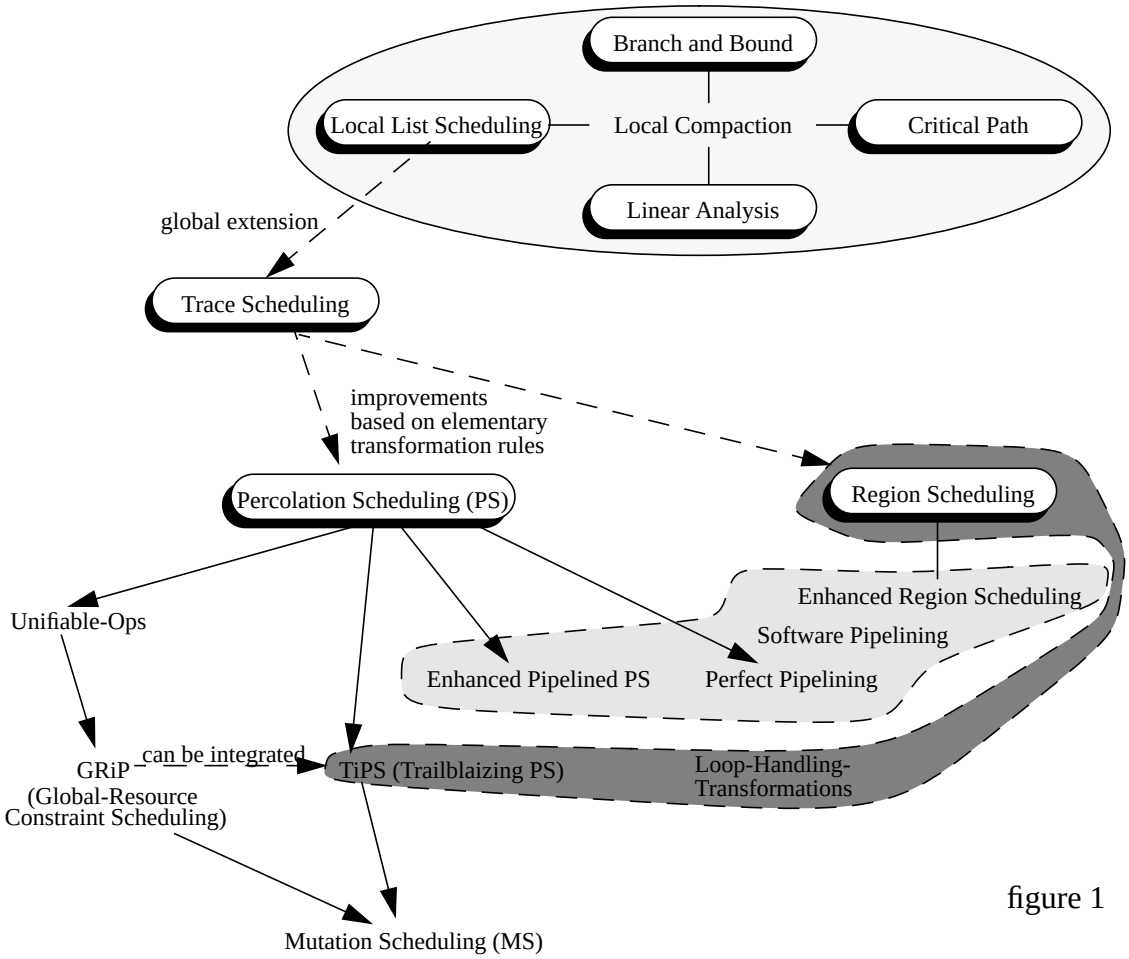

figure 1

architectures, e.g. occuring in DSP and ASIP like architectures. Encoding conflicts and restricted interconnections between machine resources have to be taken into account; retargeting, and the generation of very high quality code are very important aspects. New approaches have started to incorporate these aspects [LeMa95b, LPKS95, TSMJ95, Hein93]. However, they do not model all kinds of restrictions.

2. Contributions of this paper

Because of the mutual dependency of the three code generation phases, different techniques of phase coupling [Vegd82, NND94] has become increasingly important. Therefore, we propose a solution for scheduling, compaction and binding which is able to cooperate with code selection and register allocation. We present a formalism which is flexible and powerful enough to handle a) complicated address restrictions (even the ones described in [BaHa81]), b) all (encoding) restrictions and characteristics imposed by the controller, and c) dependencies between the instructions and the labels in order to consider unconditional and conditional jumps. The formalism is based on various types of constraints and is easily solved by a CLP system. Furthermore, the approach is incremental, i.e., the constraints can be specified at any time during code selection and allocation. These constraints are delayed until the final phase determines a schedule, but a CLP system performs local consistency checks in order to ensure the satisfiability of the set of constraints. The formalism is based on predicates and rules, which can directly be executed as a CLP program.

3. CLP: A brief overview

The constraint programming paradigm [MTP94] and especially constraint logic programming

[BeCo93] is becoming increasingly important. CLP is a conservative extension of Prolog, which allows handling constraints over various domains. We use the CLP-system ECLiPSe [ECRC95], in which all kinds of constraints can be expressed.

Definition I: Constraint

Let $V = \{X_1, ..., X_n\}$ be a finite set of variables, which take their values from their finite domains $D_1, ..., D_n$. A **constraint** $c(X_{i1}, ..., X_{ik})$ between k variables from V is a subset of the Cartesian Product $D_{i1} \times ... \times D_{ik}$.

CLP-systems come with built-in mechansims for solving, simplifying, and handling constraint sets. Satisfiability checkers support Boolean constraints and IP-solvers support linear constraints (integer or rational domain and constraints over linear terms). A CLP-system supports constraints which cannot be represented by integer programming. Prolog clauses are bidirectional and backtracking is already integrated. Extending pure Prolog with domains and constraints increases the efficiency of logic programs dramatically. Furthermore, CLP is well suited to solve problems concurrently by specifying the subproblems with constraints and solve them in one step instead of solving subproblems sequentially (e.g., the phase coupling problem in code generation [MaGo95]).

4. A CLP model for scheduling, compaction and binding

A formal description of the scheduling, compaction and binding phase follows. Since a real life example would exceed the size of this paper, we use simple examples to illustrate the formalism. First a difference between absolute code and relocatable code is made. Thereafter we define what kind of constraints are allowed to represent dependencies between instructions and labels. Next we define necessary preconditions for merging instructions, and how to merge instructions. Afterwards a set of rules is defined, describing the successive process from relocatable code to absolute code. A normal form of a relocatable program is reached, if the domain of each label is constrained to a single natural number. The last step is to replace all labels of a relocatable program by such natural numbers (constants).

We assume that a set of relocatable instructions has been generated by code selection and resource allocation. Control signals originating from the instruction memory control the datapath and the next instruction of the controller. A control signal is a part of the instruction word and activates an indivisible operation. An instruction is a set of operations which can be executed in parallel. Let AM := [Start, End] be the **address range** of the instruction memory and n its width. A **label** L_i is an integer domain variable with $Start \le L_i \le End$. An **instruction** is a bit string $\mathbf{I} \in \{0,1,X\}^n$. The value X denotes a “don’t care”. $IM_n := \{ I_n \mid I_n \in \{0,1,X\}^n \}$ with $|IM_n| \le End - Start + 1$ is a **set of instructions**. $AP_n \subset AM \times IM_n$ is an **absolute program**. A tuple $(A, I_n) \in AP_n$ represents an absolute address *A* and the corresponding unique instruction I_n.

Definition II: Relocatable instruction, set of relocatable instructions

Let V be a set of Boolean variables. A **relocatable instruction** is a tuple $RI_i = (L_i, RI_{n,i})$. L_i is a label and $RI_{n,i} \in \{\{0, 1, X\} \cup V\}^n$. $\mathbf{RIM_n} = \{(L_1, RI_{n,1}, ..., (L_m, RI_{n,m})\}$ is a **set of relocatable instructions**.

The set V is used to represent dependencies between the instruction words and the labels. Jump addresses are often coded as bit strings as part of the instruction. A 2-ary predicate *bin2dec/2* expresses the relationship between a bit string (binary value: *Bin*) and a label (decimal value: *Dec*). The predicate is bidirectional and computes binary values to decimal values and vice versa. A binary value is represented as a list *Bin*. *bin2dec/2* behaves like a constraint, if a) *Bin* is not a ground term and *Dec* is a variable or b) if *Bin* is a variable. Examples:

bin2dec([0,1,1], Label) yields Label = 3;

bin2dec([X_1, X_2, X_3], 7) yields $X_1 = X_2 = X_3 = 1$;

bin2dec([X_1, X_2, X_3, X_4], Label) is delayed, i.e., the predicate behaves as a constraint.

Assumed, the address range for jumps is restricted to even addresses, imposed by the controller. Such a restriction can be handled easily by setting the lowest bit of the binary address to 0: The constraint bin2dec([X_1, X_2, X_3, 0], Label) denotes the relationship between a binary (even) address [X_1, X_2, X_3, 0] and a decimal label.

Definition III: Relocatable Program

Let L be a set of labels and V be a set of Boolean variables. A **relocatable program** is a pair $RP_n = (RIM_n, C)$. RIM_n is a set of relocatable instructions and C is a set of linear constraints over $L \cup V$. C is used to express all dependencies between the partially ordered set of relocatable instructions (e.g., as equations or unequations). Let $(L_i, RI_{n,i})$ and $(L_j, RI_{n,j})$ be two relocatable instructions:

1. $(L_i, RI_{n,i})$ is data-dependent on $(L_j, RI_{n,j}) \rightarrow L_j < L_i$
2. $(L_i, RI_{n,i})$ is data-anti-dependent on $(L_j, RI_{n,j}) \rightarrow L_j \leq L_i$
3. $(L_i, RI_{n,i})$ is output-dependent on $(L_j, RI_{n,j}) \rightarrow L_j < L_i$

Beyond these traditionally considered dependencies [DLSM81], several other restrictions and dependencies are represented as follows:

4. $(L_i, RI_{n,i})$ must be executed in parallel to $(L_j, RI_{n,j}) \rightarrow L_j = L_i$.
5. An unconditional jump $(L_i, RI_{n,i})$ to an absolute address A is expressed by $L_i = A$.
6. An unconditional relative jump $(L_i, RI_{n,i})$ to $(L_j, RI_{n,j})$ over the relative distance D is expressed by a linear constraint $L_j = L_i + D$.
7. If a jump addresses A is coded within a substring *Bin* of the instruction word, the relationship is expressed as constraint bin2dec(Bin, A).

Example: Let $L = \{L_1, L_2, L_3, L_4, L_5\}$ be a set of labels and $V = \{X_1, X_2, X_3\}$ be a set of Boolean variables. RP_{10} is a relocatable program:

$RP_{10} = (\{ \quad (L_1, (1,0,X,1,1,0,0,1,1,X)),$
$(L_2, (1,X,X,0,X,0,X,1,0,X)),$
$(L_3, (1,X,X,0,X_1,X_2,X_3,1,0,1)),$
$(L_4, (0,X,X,0,1,0,X,1,1,0)),$
$(L_5, (0,0,X,0,1,0,0,X,1,X))\},$
$\{L_1 + 1 = L_2, L_1 < L_2, L_4 \leq L_5 , \text{bin2dec}([X_1,X_2,X_3],L_1), X_3 = 1\})$

RI_2 is data-dependent on RI_1; RI_3 is a jump instruction where the jump address $[X_1,X_2,X_3]$ is restricted to odd addresses by $X_3 = 1$ and L_1 is the corresponding decimal address; RI_5 is data-anti-dependent on RI_4; RI_2 must be an immediate successor of RI_1. Sets of linear constraints which are not simplified are allowed and possible (e.g., $L_1 + 1 = L_2$, $L_1 < L_2$).

In opposite to most previous code generators our approach is able to **merge basic block boundaries**. In certain cases it is semantically allowed, to execute a conditional jump and the first instruction of the next basic block in parallel. This is expressed by a constraint $L_i \leq L_j$, if $(L_i, RI_{n,i})$ is a conditional jump which may be executed with $(L_j, RI_{n,j})$ in parallel. Two instructions are candidates to be executed in parallel if they are compatible, i.e., no resource conflicts occur. By using a structural model of the processor including a detailed structural description of the controller, it is possible to map resource conflicts to instruction word conflicts.

Definition IV: compatible/2

Let $RI_i = (L_i, RI_{n,i})$ and $RI_j = (L_j, RI_{n,j})$ be relocatable instructions with $RI_{n,i}, RI_{n,j} \in \{\{0,1,X\} \cup$

$V\}^n$. Let $P(I_n, k)$ be the projection of a bit-string on the k-th bit (high bit on the left of a bit string, low-bit on the right, the rightmost bit position is 0). The predicate **compatible(RI_i, RI_j)** is true :⇔

$$i \neq j \wedge \forall k,\ 0 \leq k \leq n\text{-}1:$$
$$(P(RI_{n,i},k) = P(RI_{n,j},k)) \vee$$
$$(P(RI_{n,i},k) = X) \vee (P(RI_{n,j},k) = X) \vee$$
$$(P(RI_{n,i},k) \in V \wedge P(RI_{n,j},k) \notin V) \vee (P(RI_{n,j},k) \in V \wedge P(RI_{n,i},k) \notin V)$$

Example: Consider the previous example program RP_{10}: compatible(RI_2, RI_3) is true; compatible(RI_3, RI_4) is true; compatible(RI_2, RI_4) is false.

Above definition considers the fact, that two jump instructions cannot be compacted: at any bit position k at most one relocatable instruction may have a Boolean variable from V. Next, we define a function to compact two compatible instructions.

Definition V: compact: $RI_n \times RI_n \rightarrow RI_n \times CB$

Let $RI_i = (L_i, RI_{n,i})$ and $RI_j = (L_j, RI_{n,j})$ be relocatable instructions and compatible(RI_i, RI_j) is true. The function **compact($RI_{n,i}$, $RI_{n,j}$) = ($RI_{n,c}$, CB)** is defined as follows:

$$\forall k, (0 \leq k \leq n-1) : P(RI_{n,c},k) = \begin{cases} 0 \Leftrightarrow P(RI_{n,i},k) = 0 \vee P(RI_{n,j},k) = 0 \\ 1 \Leftrightarrow P(RI_{n,i},k) = 1 \vee P(RI_{n,j},k) = 1 \\ A \in V \Leftrightarrow P(RI_{n,i},k) = A \wedge P(RI_{n,j},k) = X \\ A \in V \Leftrightarrow P(RI_{n,j},k) = A \wedge P(RI_{n,i},k) = X \\ X \quad otherwise \end{cases}$$

$CB = \{A = b \mid A \in V, b \in \{0,1\}, 0 \leq k \leq n\text{-}1,$
$((P(RI_{n,i},k) = A \wedge P(RI_{n,j},k) = b) \vee (P(RI_{n,i},k) = b \wedge P(RI_{n,j},k) = A))\}$

CB is a set of bindings of Boolean variables, expressed as constraints. In a Prolog-system the function compact is simply an unification of two relocatable instructions. If the unification step fails backtracking is performed.

In the following we define rules and the preconditions which must be satisfied to apply a rule.

Definition VI: compaction rule $RP_{n,1} \rightarrow_c RP_{n,c}$

Let $RP_n = (RIM_{n,1}, C)$, $RP'_n = (RIM'_n, C_c)$ be relocatable programs, $RI_i = (L_i, RI_{n,i}) \in RP_n$, $RI_j = (L_j, RI_{n,j}) \in RP_n$ and compatible(RI_i,RI_j). Then the **compaction rule** $RP_n \rightarrow_c RP'_n$ can be applied:⇔

1. $(RI'_n, CB) := compact(RI_{n,i}, RI_{n,j})$
2. $RIM'_n := RIM_n \setminus \{(L_i, RI_{n,i}), (L_j, RI_{n,j})\} \cup \{(L_i, RI_{n,c})\}$
3. $C_c := C \cup \{L_i = L_j\} \cup CB$
4. C_c is a consistent set of linear constraints

Example: RI_2 and RI_3 of RP_{10} are compatible, i.e., we can apply the function compact. In a first step, RP_{10} can be replaced by RP'_{10}:

$RP'_{10} = (\{ (L_1, (1,0,X,1,1,0,0,1,1,X)),$
$(L_2, (1,X,X,0,0,0,1,1,0,1)),$
$(L_4, (0,X,X,0,1,0,X,1,1,0)),$
$(L_5, (0,0,X,0,1,0,0,X,1,X))\},$

$\{L_1+1 = L_2, L_1 < L_2, L_4 \leq L_5 , bin2dec([X_1,X_2,X_3],L_1) , X_1 = 0, X_2 =0 , X_3 = 1, L_2 = L_3 \})$

The jump address $[X_1,X_2,X_3]$ has been bound to [0, 0, 1], i.e., $L_1 = 1$, $L_2 = 2$. RI_4 and RI_5 may be compacted in the next step. In a CLP-system the set of constraints is automatically simplified, e.g., by eliminating constraints in which all arguments are instantiated. The compaction rule reduces the set of relocatable instructions and the domain of the labels step by step.

Definition VII: selectVar/2, selectValue/2

Let $V = \{X_1, ..., X_m\}$ be a finite set of variables with associated finite domains $D_1, ..., D_m$. The predicate **selectVar(V, X_i)** is true :$\Leftrightarrow X_i \in V \wedge |D_i| > 1$ $(1 \leq i \leq m)$. The predicate **selectValue(X_i, V_i)** is true :$\Leftrightarrow V_i \in D_i$.

This simple definition is very useful in a CLP-system. By *selectVar/*2 we can select a variable X_i to be instantiated with *selectValue/*2 by a value V_i out of its domain D_i. If the domain of a variable has only one element ($|D_i| = 1$), the variable is automatically bound to this value. Several heuristics can be integrated: E.g., the most constrained variable or the variable with the smallest/largest domain can be selected. This search process in finite domains is called **labelling**.

Let $S_1 \rightarrow S_2$ be a rule which rewrites a state S_1 by a state S_2. $S_1 \rightarrow^* S_n$ denotes a chain of applications of this rule ($\rightarrow^*$ denotes the transitive and reflexive closure). Usually a relocatable program RP_n has several successor programs RP'_n, which can be derived from the compaction rule. However, the rules are not confluent. Therefore, the backtracking mechanism of a CLP-system is exploited. After applying the compaction rule several times, the relocatable program cannot be compacted any more. Now a labelling rule has to be applied.

Definition VIII: labelling rule $RP_{n,c} \rightarrow_l RP_{n,l}$

Let RP_n, $RP_{n,t}$, $RP_{n,c} = (RIM_{n,c},C_c)$ and $RP_{n,l} = (RIM_{n,l}, C_l)$ be relocatable programs. $RP_n \rightarrow_c^* RP_{n,c}$ is a chain of applications of the compaction rule and the following holds: $\neg\exists\ RP_{n,t}: RP_{n,c} \rightarrow_c RP_{n,t}$. $LV := L \cup V$ is the union of the labels and the Boolean variables of $RP_{n,c}$.
The l**abelling rule** $RP_{n,c} \rightarrow_l RP_{n,l}$ can be applied :$\Leftrightarrow$

1. $RIM_{n,l} := RIM_{n,c}$
2. selectVar(LV, X_i)
3. selectValue(X_i, V_i)
4. $C_l := C_c \cup \{X_i = V_i\}$
5. C_l is a consistent set of linear constraints

If the labelling rule is applied, it selects for exactly one domain variable a value out of the domain. A chain of applications of the labelling rule selects values for all variables. Afterwards, the resulting program has to be syntactically transformed into an absolute program. This is done by the last rule:

Definition IX: bind: $RP_n \rightarrow AP_n$

RP_n, $RP_{n,t}$ and $RP_{n,l} = (RIM_{n,l}, C_l)$ are relocatable programs and AP_n is an absolute program. L is the set of labels of $RP_{n,l}$. $RP_n \rightarrow_l^* RP_{n,l}$ is a chain of applications of the of the labelling rule, such that it cannot be applied again to $RP_{n,l}$: $\neg\exists\ RP_{n,t}: RP_{n,l} \rightarrow_l RP_{n,t}$. The function **bind($RP_{n,l}$)** = AP_n is defined as follows: $AP_n := \{ (V_i, RI_{n,i}) \mid (L_i, RI_{n,i}) \in RIM_{n,l} \wedge \exists C_i \in C_l: L_i = V_i) \}$

Example: The resulting semantically equivalent absolute program for RP_{10} is:

AP_{10} = ({ (1, (1,0,X,1,1,0,0,1,1,X)),
(2, (1,X,X,0,0,0,1,1,0,1)),
(3, (0,0,X,0,1,0,0,1,1,0)) }

Labels L_4 and L_5 have been bound to address 3 in order to take the autoincrement operation of the

program counter into account (if the instruction is not a conditional or unconditional jump).

The termination of scheduling, compaction and binding is guaranteed, because the above defined rules derives a more and more restrictive set of constraints. It is not guaranteed, that the above presented approach generates an optimal solution, due to the NP-hardness of optimal code compaction. The set of rules have quadratical run time, because at most all pairs of relocatable instructions can be applied by the compaction rule. The formalism can be easily extended to get an optimal solution. With a minimum of programming effort, we could use a ECLiPSe built-in predicate to search for an optimal solution for an objective function (number of code lines). Furthermore, it is possible to perform an optional compaction, i.e., to perform just scheduling and binding without exploiting the parallelism of the target architecture. This is possible by specifying simply an additional constraint *alldifferent(L)* for the set of labels L. Hereby, it is requested that all labels have different values. Such a constraint is an ECLiPSe built-in predicate.

5. Results

The presented formalism is implemented in **RESTART** [BiMa95, Biek95a, Biek95b] (retargetable compilation of self-test programs using constraint logic programming). RESTART has been successfully used to compile self-test programs for various processors. However, the presented CLP model is general enough to be integrated in any retargetable code generator. The complete scheduling, compaction and binding phase consists of about 350 lines of ECLiPSe code. Table 1 describes some example circuits: the general purpose microprocessors SIMPLECPU [BiNe96], DEMOCPU [Biek95], MAC-1 [Tane90], REF [MIMO94] and manocpu [Mano93]; prips [ABMN93] is a coprocessor with a RISC-like instruction set, which provides data types and instructions supporting the execution of Prolog programs. The processor has to be described at the register transfer (RT) level (MIMOLA [MIMO94] or VHDL). The number of RTL components, the width of the datapath, the width of the microinstruction controller, the number of registers, the number of arithmetical logical units and the length of the hardware description in lines is given.

Table 1: Target architectures

processor	#RT	data path	controller	#registers	#ALUs	#HDL-lines
SIMPLECPU	10	4	20	16	1	275
DEMOCPU	11	4	18	16	1	273
MAC-1	14	16	32	18	2	380
REF	16	16	84	16	3	349
MANOCPU	21	16	50	9	1	531
PRIPS	50	32	83	37	8	1290

Table 2 shows results for the total retargetable code generation process and the scheduling, compaction and binding phase. The number of generated (micro-) instructions (#I), CPU time in seconds for the total retargetable code generation process (sec. total), time for scheduling, compaction and binding in seconds (scb.time) and the ratio #I/scb.time is given. All times are measured on a SPARC 20 workstation. Different (self-test) programs have been compiled. Of course, we can not expect a retargetable compiler to be as fast as a usual (target dependent) compiler.

6. Conclusions

We presented a flexible and powerful formalism for scheduling, compaction and binding which is

Table 2: Results: Retargetable code generation; scheduling, compaction and binding

processor	#I	sec. total	scb.time	#I/scb.time
SIMPLECPU	54	4.7	1.53	35.3
DEMOCPU	31	2.6	0.55	56.3
MAC-1	78	57.9	6.23	12.5
REF	108	35.7	9.53	11.3
MANOCPU	122	43	10.23	11.9
PRIPS	35	69.3	4.8	7.3

directly executable in a CLP-system. The incremental approach can be coupled with the other code generation phases, namely code selection and resource allocation. It is intended to be used in a retargetable code generator and is integrated in RESTART, a retargetable compiler for self-test programs. A CLP-system handles a set of constraints concurrently to the execution of other tasks and inconsistencies can be immediately detected in order to find the best backtracking entry point. A good strategy to avoid wrong decisions during a search process is to delay the decisions as long as possible. This is done by constraints and the CLP-inherent delay mechanism (coroutining). Furthermore, the presented formalism is capable of handling different controller pecularities, restrictions and address generation schemes.

7. References

[ABMN93] C. Albrecht, S. Bashford, P. Marwedel, A. Neumann, W. Schenk. The design of the PRIPS microprocessor, 4th EUROCHIP-Workshop on VLSI Training, Toledo, 1993.

[AiNi88] A. Aiken, A. Nicolau. Perfect Pipelining: A new Loop Parallelization Technique. European Symposium on Programming, Springer LNCS 300, 1988.

[AJLS92] V.H. Allan, J. Janardhan, R.M. Lee and M. Srinivas. Enhanced Region Scheduling on a Program Dependence Graph. MICRO-25, pp. 72-80, 1992.

[BaHa81] T. Baba, H. Hagiwara. The MPG System: A Machine-Independent Efficient Microprogram Generator. IEEE Trans. on Computers, Vol. C-30, pp. 373 - 395, June 1981.

[Beat91] S.J. Beaty. Instruction Scheduling Using Genetic Algorithms. Ph.D. Thesis, Department of Mechanical Engineering, Colorado State University, Fort Collins, Colorado, 1991.

[BeCo93] F. Benhamou, A. Colmerauer (eds.). Constraint Logic Programming. Selected Research. MIT Press, Cambridge, London, 1993.

[BGS95] D.A. Berson, R. Gupta, M.L. Soffa. GURRR: A Global Unified Resource Requirements Representation. SIGPLAN Notices, Proceedings of the ACM SIGPLAN on Intermediate Representations IR'95, Vol. 30(4), pp. 23-34, San Francisco, California USA, April 1995, http://www.cs.pitt.edu/~berson/papers.html.

[Biek95a] U. Bieker. Retargetable Compilation of Self-Test Programs Using Constraint Logic Programming. In [MaGo95], May 1995.

[Biek95b] U. Bieker. Retargierbare Compilierung von Selbsttestprogrammen digitaler Prozessoren mittels Constraint-logischer Programmierung. Dissertation, Shaker Verlag, Aachen, ISBN 3-8265-10925, December 1995.

[BiMa95] U. Bieker, P. Marwedel. Retargetable Self-Test Program Generation Using Constraint Logic Programming. 32nd Design Automation Conference, pp. 605 - 611, San Francisco, June 1995.

[BiNe96] U. Bieker, A. Neumann. Using Logic Programming and Coroutining for electronic CAD. Journal of Logic Programming, February 1996.

[DaTa76] S. Dasgupta, J. Tartar. The Identification of Maximal Parallelism in Straight-line Microcode. IEEE Transactions on Computers, Vol. C-25(10), pp. 986-992, October 1976.

[DLSM81] S. Davidson, D. Landskov, B. Shriver, P. Mallett. Some Experiments in Local Microcode Compaction for Horizontal Machines. IEEE Trans. on Computers, Vol. C-30, pp. 460 - 477, July 1981.

[EbNa89] K. Ebcioglu and T. Nakatani. A new Compilation Technique for Parallelizing Loops with Unpredictable Branches. 2nd Workshop on Programming Languages and Compilers for Parallel Computing, 1989.

[EbNi89] K. Ebcioglu, A. Nicolau. A Global Resource Constrained Parallelization Technique.Proceedings of the 2nd International Conference on Supercomputing, pp. 154-163, 1989.

[ECRC95] ECLIPSE 3.5 User Manual. ECRC Common Logic Programming System. ECRC GmbH, Arabellastr. 17, München, 1995.

[Faut95] A. Fauth. Beyond Tool-Specific Machine Descriptions. In [MaGo95], 1995.

[Fish81] J.A. Fisher. Trace Scheduling: A Technique for Global Microcode Compaction. IEEE Transactions on Computers, Vol. C-30(7), pp. 478-490, July, 1981.

[Gasp89] F. Gasperoni. Compilation Techniques for VLIW Architectures. Ph.D. Thesis, Courant Institute of Mathemathical Science, New York University, March, 1998.

[GuSo90] R. Gupta, M.L. Soffa. Region Scheduling: An Aproach for Detecting and Redistributing Parallelism. IEEE Trans. on Software Engineering, Vol. 16(4), pp. 421-431, April, 1990.

[Hein93] W. Heinrich. Formal Description of Parallel Computer Architectures as a Basis of Optimizing Code Generation. Ph.D. Thesis, Institut für Informatik, TU München, 1993.

[LDSM80] D. Landskov, S. Davidson, B.D. Shriver, P.W. Mallet. Local Microcode Compaction Techniques. ACM Computing Surveys, Vol. 12(3), pp. 261-294, 1980.

[LeMa94] R. Leupers, M. Marwedel. Instruction Set Extraction from Programmable Structures. EURO-DAC, Grenoble, pp. 156 - 161, September 1994.

[LeMa95] R. Leupers, P. Marwedel.Time-constrained Code Compaction for DSPs. Int. Symp. on System Synthesis (ISSS), September 1995.

[LPKS95] D. Lanneer, J. V. Praet, A. Kifli, K. Schoofs, W. Geurts, F. Thoen, G. Goossens. CHESS: Retargetable Code Generation for Embedded DSP Processors. In [MaGo95], 1995.

[MaGo95] P. Marwedel, G. Goossens (eds.). Code Generation for Embedded Processors, Kluwer Academic Publishers, London, 1995.

[Mano93] M. Morris Mano. Computer System Architecture. Prentice-Hall International, Inc., Third Edition, 1993.

[MIMO94] S. Bashford, U. Bieker, B. Harking, R. Leupers, P. Marwedel, A. Neumann, D. Voggenauer. The MIMOLA Language - Version 4.1. Technical Report, Computer Science Dept., University of Dortmund, September 1994.

[MoEb92] S. Moon, K. Ebcioglu. An Efficient Resource Constraint Global Scheduling Technique for Superscalar and VLIW Processors . MICRO-25, pp. 55-71, December, 1992.

[MTP94] B. Mayoh, E. Tyugu, J. Penjam (eds.). Constraint Programming, Springer, Berlin, 1994.

[Nico85] A. Nicolau. Percolation Scheduling: A Parallel Compilation Technique. Ph.D. Thesis, Department of Computer Science, Cornell University,Ithaca, New York, May, 1985.

[NoNi92] S. Novack, A. Nicolau. An Efficient Global Resource Constrained Technique for Exploiting Instruction Level Parallelism. Proceedings of the International Conference on Parallel Processing. pp. II 297-301, August 1992.

NoNi93] S. Novack, A. Nicolau. Trailblaizing: A Hierarchical Approach to Percolation Scheduling. Irvine University, Technical Repoert TR-92-56, August 1993.

[Nowa87] L. Nowak. Graph Based Retargetable Microcode Compilation in the MIMOLA Design System. Proc. of the 20th An. Workshop on Microprogramming (MICRO-20), pp. 126 - 132, 1987.

[NND94] S. Novack, A. Nicolau, N. Dutt. A Unified Code Generation Approach using Mutation Scheduling. Technical Report 94-35, University of California, Irvine, 1994.

[PLMS95] P. G. Paulin, C. Liem, T. C. May, S. Sutarwala. Flexware: A Flexible Firmware Development Environment for Embedded Systems. In [MaGo95], 1995.

[Rama74] C.V. Ramamoorthy, M. Tsuchiya. A High-Level Language for Horizontal Microprogramming. IEEE Transactions on Computers, Vol. C-23(8), pp. 791-801, August 1974.

[Tane90] A. Tanenbaum. Structured Computer Organization. 3. Edition. Prentice-Hall, Inc., 1990.

[TSMJ95] A. H. Timmer, M. T. J. Strik, J. L. van Meerbergen, J. A. G. Jess. Conflict Modelling and Instruction Scheduling in Code Generation for In-House DSP Cores. Proceedings of the 32nd Design Automation Conference, pp. 593 - 598, San Francisco, June 1995.

[Vegd82] S. R. Vegdahl. Phase Coupling and Constant Generation in an Optimizing Microcode Compiler. MICRO-15, pp. 125 - 133, Palo Alto, October 1982.

[YST74] S.S. Yau, A.C. Schowe, M. Tsuchhiya. On Storage Optimization of Horizontal Microprograms, MICRO-7, pp. 98-106, 1974.