# Constraint Driven Code Selection for Fixed-Point DSPs

Steven Bashford, Rainer Leupers*
Dept. of Computer Science 12
University of Dortmund, Germany
email: bashford@ls12.cs.uni-dortmund.de

**Abstract– Fixed-point DSPs are a class of embedded processors with highly irregular architectures. This irregularity makes it difficult to generate high-quality machine code from programming languages such as C. In this paper we present a novel constraint driven approach to code selection for irregular processor architectures, which provides a twofold improvement of earlier work. First, it handles complete data flow graphs instead of trees and thereby generates better code in presence of common subexpressions. Second, the presented technique is not restricted to computation of a single solution, but it generates alternative solutions. This feature enables the tight coupling of different code generation phases, resulting in better exploitation of instruction-level parallelism. Experimental results indicate that our technique is capable of generating machine code that competes well with hand-written assembly code.**

## 1 Introduction

Today, many embedded systems employ programmable processors as their core components. The machine code running on embedded processors frequently must meet tight speed and size constraints. This is due to the presence of real-time requirements and limited silicon area for program memories. These requirements frequently prevent the use of high-level language compilers for embedded software development, since compilers usually cause an overhead in code quality as compared to hand-written assembly code.
In this paper, we consider fixed-point DSPs as a specific class of embedded processors. In particular for this type of processors compilers tend to produce an intolerably large overhead in code size and performance [16]. As a consequence, the largest part of fixed-point DSP software is still written manually in assembly languages. This implies a bottleneck in system development and also reduces some of the advantages of embedded software (as compared to ASIC hardware), such as reusability and portability. Our overall goal is to eliminate this bottleneck by providing better code generation techniques for high-level language compilers, which take the peculiarities of fixed-point DSPs into account.

The poor quality of compiler-generated code for fixed-point DSPs is primarily caused by the irregular architecture of such processors, which in turn is a consequence of the demand for very efficient processors in the DSP area. By "irregularity" of the processor architecture we denote the following features: *Special-purpose registers* may not be orthogonally accessible by all functional units, but may be connected to the inputs and outputs of specific functional units. There may be *chained operations*, where the most important example in DSPs is the MAC (multiply-accumulate) instruction. Furthermore, fixed-point DSPs typically show *restricted instruction-level parallelism*, i.e., they do permit the parallel execution of several instructions per instruction cycle, but unlike in VLIW machines, the permissible combinations of parallel instructions are quite restricted.

In this contribution we focus on the task of *code selection* for fixed-point DSPs or, more generally, for processor architectures with irregular data paths. Code selection is concerned with mapping an intermediate representation (IR) of the source program to machine instructions of the target processor. This task can be viewed as "covering" the IR by machine instruction patterns. Most current code selection techniques are based on *tree covering* and operate on data flow tree (DFT) based IRs of basic blocks. However, tree covering in general produces suboptimal covers for basic blocks. Since basic blocks generally appear in the form of data flow graphs (DFGs), DFGs have to be split into DFTs (fig. 1). This is performed by cutting DFGs at nodes representing multiple uses of values (common subexpressions, CSEs).

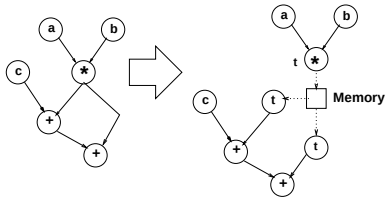


Figure 1: *Splitting a DFG into DFTs*

As we will discuss later, the tree-based approach has drawbacks particularly for irregular architectures and leads to inferior code quality. In this contribution we therefore propose a novel code selection technique, which generalizes code selection for irregular processor architectures from DFTs to DFGs,

*The authors acknowledge the support by the DFG and HP EESof


DAC 99, New Orleans, Louisiana

thereby producing more efficient machine code. This code selection technique is capable of covering complete DFGs by machine instructions and is based on the paradigm of constraint logic programming (CLP).
The paper is structured as follows: The next section provides a discussion of the limitations of DFT covering based code selection for fixed-point DSPs and also mentions related work. Section 3 shortly sketches concepts of constraint logic programming. In section 4 a model for representing alternative DFG covers and the DFG covering approach itself are presented. Section 5 describes several applications of the DFG covering technique in a compiler and provides experimental results indicating the code quality improvements achieved.

# 2 Motivation and related work

Throughout this paper we will represent processor operations by register transfers (RTs), that reflect the operations performed and the storage resource (SR) locations for the operands and the result. In fig. 2 a partial data path of an Analog Devices ADSP-210x together with a subset of its RTs are shown. For instance, in RT $ar := ax + ay$ the first and second operand must reside in SRs *ax* and *ay*, respectively. The result is stored in SR *ar*. In the following, we call *ar* the *definition* of the RT and *ax*, *ay* its *uses*. We say $use_i$, when referring to operand number *i*.

## 2.1 Drawbacks of DFT covering

Most of today's code selection approaches are based on tree covering or tree parsing. Covering consists of mapping DFT nodes to available RTs. DFT edges may be mapped to RTs denoting pure transfer (move) operations, required for routing data between the functional units (FUs). An example of a cover is shown in fig. 3 a. Tree covering is typically based on *tree pattern matching* combined with *dynamic programming*. Tree pattern matchers are used to determine the set of alternative covers for a DFT. Dynamic programming serves to select the optimal cover from the alternatives. Such code generators can be generated automatically by tools like iburg [3], which require a formal description of the target instruction set given as a tree grammar.
For applying tree covering, DFGs have to be split into DFTs (fig. 1), where uses of CSEs are represented as variable nodes with the same identifier. Tree covering requires that these variable nodes are mapped to a certain fixed SR (typically the memory) in advance.

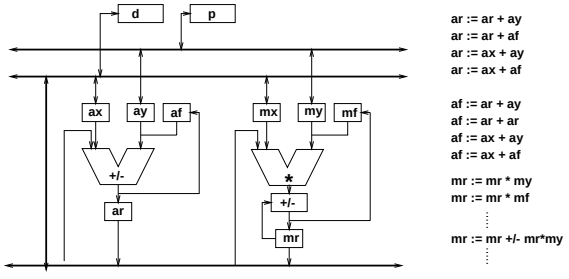


Figure 2: *Partial ADSP-210x data path with RTs*

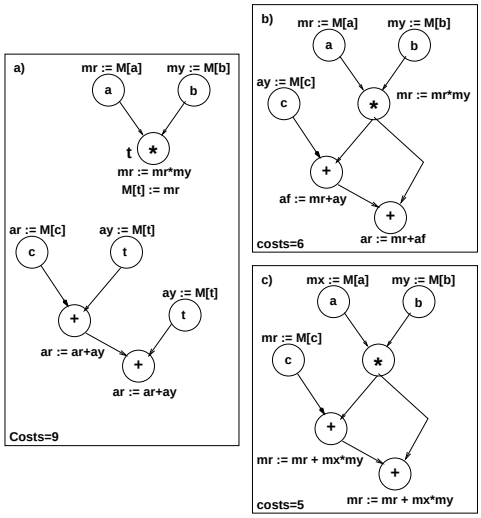


Figure 3: *DFT vs. DFT covering*

As an example consider fig. 3 a), where variables are mapped to memory. Thus, the CSE labeled with *t* is moved from SR *mr* to memory, and each use loads *t* back into SR *ay*. If code selection was done for the complete DFG, then the overall costs of the cover (i.e. the number of instructions required to implement the DFG) could be reduced from 9 to 6 (fig. 3 b). Note, that not only the location of the definition and the uses of the CSE change, but also the complete covering of the second DFT (exploiting commutativity of operators). Fig. 3 c) shows the additional optimization effect achieved by allowing CSEs to occur as sub-operations in chained operations, which again decreases the costs by one instruction.

A further drawback of approaches based on tree covering is that *instruction-level parallelism* (ILP) cannot be taken into account during covering. Fig. 4 shows two covers with the same costs, but only the second cover can be mapped to parallel code for the ADSP-210x: Parallel transfer operations are only feasible if one operand is moved from memory *d* to *mx*, and the second one from *p* to *my*. Therefore, it is obviously favorable to keep alternative covers of the same costs for the scheduling phase, during which the most appropriate alternative may be selected.

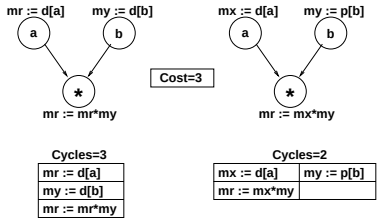


Figure 4: *Impact of covers on ILP*

In the following, we discuss related work, which is based on either tree covering or a phase-coupled approach to DFG code generation.

## 2.2 Approaches based on tree covering

In the CBC and RECORD compilers [2, 7], processor models are mapped to iburg [3] specifications, from which code generators are automatically generated. In [1] DFGs are transformed into DFTs by pruning edges of CSEs based on the "RTG criterion", leading to larger DFTs. Covering is then performed with help of the code generator generator olive (an extension of iburg). However, the RTG criterion only holds for a very specific class of fixed-point DSPs. In contrast to these approaches our technique enables optimal DFG covering (instead of DFTs), simultaneously taking into account routing costs of CSEs and allowing CSEs to be mapped to chained operations.

## 2.3 Approaches based on phase coupling

Several approaches are based on the paradigm of coupling different code generation phases (including code selection), so as to maximize code quality. In [15, 4, 5, 9] optimal code is generated for DFGs, taking into account register allocation and instruction scheduling. Code generation phases are described in the form of constraints (generally linear equations and inequations). The complete solution space is explored while all constraints are considered simultaneously. Generally only basic blocks of very limited size and a restricted set of architectures can be handled by this approach. Other phase coupling approaches are based on heuristics and postpone the final selection of a certain RT to register allocation, and/or scheduling [12, 13, 6, 8]. In each step during register allocation or instruction scheduling, resources are strictly bound.

In contrast to these approaches, the presented technique allows to choose between optimal and heuristic (for large DFGs) DFG covering. More important, our technique does not generate only a single DFG cover, but retains alternative DFG covers for later code generation phases. Alternatives are kept as long as possible, leaving much freedom for decisions during the scheduling phase.

# 3 Constraint logic programming

Our approach is implemented in the *constraint logic programming* (CLP [10]) language ECLiPSe[1] [14]. The modeling technique of our covering approach is based on *constraint satisfaction problems* (CSPs). CSPs are represented by a set of variables and a set of constraints, describing dependencies between the variables. Variables are associated with certain domains (i.e., sets of values), therefore also called *domain variables*. A CSP solution is a mapping of each variable to a certain member of its domain, which meets all constraints. The goal is to find one valid solution or an optimal solution according to an objective function.

A basic technique used in CLP systems is *constraint propagation*. Constraints locally check the feasibility of the domains, and they also eliminate infeasible members of domains. In ECLiPSe, constraints act like agents in the background and may be either in a woken or suspended state.

[1] ECLiPSe is based on PROLOG and comes along with a set of domains together with dedicated constraints, solvers, and search and optimization strategies.

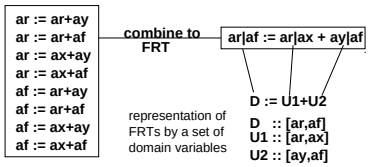


Figure 5: *Representation of multiple RTs by a FRT*

Waking of constraints is performed automatically by the system, e.g. if the domains of a variables are altered. As an example we consider the variables $X$ and $Y$, both associated with the domain $\{1,..,10\}$. If we now impose the constraint $X < Y$, the domain of $X$ is reduced to $\{1,..,9\}$, because there is no legal assignment of $Y$ meeting the constraint, if $X$ is set to 10. As there is no feasible solution if $Y = 1$, the domain of $Y$ is reduced to $\{2,..,10\}$. A setting of $X$ to 5 leads to a reduction of $Y's$ domain to $\{6,..,10\}$, caused by the reactivation of $X < Y$. Solving is based on search strategies by means of constraining the variables to certain members of their domains, which is denoted as *labeling*. Given a set of variables $V$, a certain labeling strategy defines an order for traversing the variables and a strategy for selecting members from the domains. The constraints guide the labeling of variables. Constraint propagation serves for pruning the search space in each labeling step by reducing the domains of unlabeled variables and leads to an early detection of the infeasibility of a partial labeling. Any labeling which does not meet the given set of constraints is rejected and leads to backtracking. ECLiPSe provides several predefined labeling strategies, but also enables user defined labeling strategies.

For optimization problems, there are predefined generic optimization procedures based on branch and bound strategies. Given a set of variables[2] $V$, the optimization procedures expect a labeling strategy $l(V)$ and an objective function $cost(V)$ defining the costs $C$. Solving the optimization problem is performed by calling one of the predefined optimization procedures in the form of $optimize(l(V), cost(V))$. Each time a new minimal solution $C'$ is found, a new constraint $C < C'$ is added and search is continued, triggered by the backtracking mechanism of ECLiPSe. Given an appropriate design of the model, the efficiency for finding solutions basically depends on the labeling strategy, which may be specified by the user.

# 4 Generation of DFG covers

An essential point in our code selection methodology is a new representation of DFG covers and the computation of the set of all alternative covers of a DFG by means of a set of domain variables and constraints. We will first introduce our model of alternative DFG covers, then describe the generation of all alternative covers, and conclude this section with runtime results.

## 4.1 Model of alternative DFG covers

To represent alternative DFG covers, the resources of all RTs matching a node in the DFG are combined to a *factorized*

[2] This may also be any data structure $V$, containing domain variables.

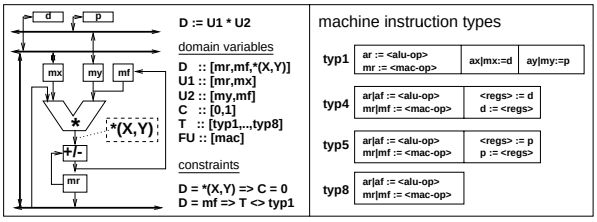


Figure 6: *Representation of FRTs*

*RT* (FRT). The resulting FRT is represented by a set of domain variables, representing alternative resource sets. In fig. 5 RTs of the `"+"` operation of the ADSP-210x and a subset of the domain variables of the resulting FRT are shown (a domain $\{v_1,..,v_n\}$ is associated with variables $X$ by writing $X :: [v_1,...,v_n]$). The set of possible RTs is now given by the possibilities of combining the resources. Restrictions on combining certain resources are expressed in the form of constraints.

A complete definition of a FRT is given by the tuple $(Op, D, [U_1,\ldots,U_n], F, C, T, CS)$. $Op$ denotes the operation. The domain variables $D$ and $U_1,\ldots,U_n$ represent the alternative SR locations for the definition and the uses. $F, T$, and $C$ denote the *extended resource information* (ERI), specifying the used FU $F$, the costs $C$ (given as the number of instruction cycles required to execute the RT), and a machine instruction type $T$. Machine instruction types specify, how RTs can be combined to machine instructions, in order to be executed in parallel. A subset of the machine instruction types of the ADSP-210x are shown in fig. 6. Types and FUs are used to model potential parallelism between RTs. $CS$ is the constraint set defining the mutual dependencies between the SRs. It may also specify dependencies between SRs and ERIs. Thus, effects like selecting certain FUs and machine types (e.g., during scheduling) are also tightly coupled with the covering process.

As an example consider the set of RTs $\{c := a+b, a := c+b\}$. The FRT is described by $(+, D, [U_1,U_2], F, C, T, \{D \in \{c,a\}, U_1 \in \{c,a\}, U_2 = b\})$. Note, that the specification of domains is also given in the form of constraints. Additionally, we need constraints to describe the dependency between $D$ and $U_1$. This can be expressed by the constraint $D = c \leftrightarrow U_1 = a$. If we now set $D$ to $c$, the constraint causes the reduction of $U_1$ to $a$. As a further example, we consider the FRT specification for the operation `"*"` of the ADSP-210x in fig. 6. The definition $D$ consists of the alternative SRs $mr$ and $mf$, and a *virtual SR* (VSR) $*(X,Y)$. This VSR takes into account, that the multiplication may also be a sub-operation of the chained MAC operation (e.g., $mr|mf := mr + mr|mx * my|mf$). If $D$ is set to $*(X,Y)$, the multiplication is part of the MAC operation. Then, the costs are set to zero ($C = 0$), since the costs of the MAC operation are associated with the node denoting the `"+/-"` operation. The constraint $D = mf \rightarrow T \neq typ1$ takes ILP conditions into account: If $D$ is located to $mf$, $typ1$ is eliminated from $T$'s domain. During scheduling, only FRTs with equal types can be assigned to the same machine instruction.

The information to generate FRTs are held in the *internal machine model* of the code generator. The covering process accesses information of the internal machine model via the interface predicate $match(Op, D, [U_1,\ldots,U_n], F, C, T)$. The FRT which matches the operation $Op$ associates domains and constraints with the input variables of *match*. Since constraints are handled internally by the CLP system, they are not passed as an argument to *match*. The internal machine model also provides constraints to ensure the existence of transfer paths between definitions and uses of values. The interface predicate $\rightarrow^* (D, U, C)$ holds if there exists a path from a definition $D$ to a use $U$, and also comprises the dependencies between locations $D, U$ and transfer costs $C$. On the ADSP-210x, there is no transfer path from $mf$ to any other SR. We denote such SRs as *deadlock SRs*. Furthermore, locating a definition to a VSR always requires to also locate all uses to the same VSR. Thus, $\rightarrow^* (D, U, C)$ is expressed by $(D = mf \leftrightarrow U = mf) \wedge (D = *(X,Y) \leftrightarrow U = *(X,Y))$.

## 4.2 The FRT covering process

Our code selection approach covers each DFG node with a FRT. In the following, $n_i$ will denote the $i$-th node in a DFG. The FRT associated with $n_i$ is denoted as $frt_i$. $D_i$ denotes the definition of $frt_i$ and $U_{i,j}$ denotes the $use_j$ of $frt_i$. The function $vd(i,j)$ returns $k$, where $n_k$ defines the value used by $U_{i,j}$. Each $n_i$ is also associated with variables $TC_{i,j}$, denoting the transfer costs for each $U_{i,j}$. There is a predicate $cse(i)$, which holds if $n_i$ is a CSE.

For a given DFG, FRT covering is performed by applying the constraint *match* to each $frt_i$, which yields a new instance of the matching FRT in the internal machine model. For each $use_j$ of $frt_i$ we apply the transfer constraint $\rightarrow^* (D_{vd(i,j)}, U_{i,j}, TC_{i,j})$, in order to ensure the existence of a path between each definition $D_{vd(i,j)}$ to its use $U_{i,j}$. With each node representing a program variable, a set of possible initial locations at the beginning of a basic block is associated with the definition $D_i$. With each common subexpression $CSE_i$ we associate extra variables $D'_i$ and $TC'_i$. Covering of common subexpressions rooted at $n_i$ is handled by additionally defining $\rightarrow^* (D_i, D'_i, TC'_i)$. This allows to define extra locations for CSEs. In order to keep the model more simple, we assume, that every node has these extra variables, and declare constraints $D'_i = D_i \wedge TC'_i = 0$ for each non-CSE node. The approach of FRT covering can be declaratively stated as:

$$\forall n_i : match(frt_i) \wedge \forall U_{i,j} \in frt_i :\rightarrow^* (D'_{vd(i,j)}, U_{i,j}, TC_{i,j})$$

$$\forall n_i \wedge cse(i) :\rightarrow^* (D_i, D'_i, TC'_i)$$

| source | runtime | nodes | edges | CSEs |
|---|---|---|---|---|
| exm | 0.28 | 6 | 7 | 1 |
| iir | 0.62 | 17 | 19 | 2 |
| cu | 0.96 | 17 | 18 | 4 |
| cm | 0.85 | 13 | 14 | 4 |
| lf | 1.52 | 23 | 27 | 8 |
| t1 | 2.41 | 24 | 38 | 8 |
| t2 | 3.04 | 29 | 47 | 7 |
| t3 | 2.15 | 21 | 33 | 9 |
| t4 | 7.99 | 82 | 153 | 15 |

Table 1: *Runtimes for FRT covering*

In table 1 the runtimes of FRT covering for several example DFGs are shown. These examples comprise the one from fig. 3, some DSPStone benchmarks [16] (complex multiply (cm), complex update (cu), iir filter (iir)), a lattice filter (lf)

and some internal benchmarks (t1-t4). All runtimes in this paper are given in UltraSparc CPU seconds. The runtime data indicate that FRT covering is efficient. The table also shows some characteristics of the benchmarks: the number of DFG nodes, edges and CSEs. One can show that the worst case complexity of FRT covering is $O(N * D^2)$, where $N$ is the number of DFG nodes, and $D$ is the maximum number of either FUs, SRs, or instruction types. An important feature of our approach is, that a generally exponential number of alternative covers is stored in a representation of linear size (w.r.t. to the number of DFG nodes $N$), by means of FRT covers.

# 5 Applications of FRT covering

This section describes several applications of the presented FRT covering technique in code selection and code generation for DFGs. First, we consider optimal code selection for DFGs with respect to a sequential instruction execution model, i.e., neglecting ILP. We show the improvements in code quality achieved as compared to a tree-based code selection approach. Since optimal DFG covering may be too computation-time intensive for large DFGs, we also present a heuristic variant, which achieves close-to-optimum results within reasonable computation times. After that, we briefly describe the integration of the code selection phase into a phase-coupled code generation technique, where alternative DFG covers are exploited to maximize ILP, resulting in very compact machine code.

In the following we make use of the following notations: for $n_i$, the set $CV_i = \{C_i, TC_i'\} \cup \{T_{i,j} | U_{i,j} \in frt_i\}$ denotes the cost variables; $Cost_i = \sum_{c \in CV_i} c$ denotes the cover costs, and $SR_i = \{D_i, D_i'\} \cup \{U_{i,j} | U_{i,j} \in frt_i\}$ is the set of SR locations of definitions and uses. We will denote the root node of a $DFT_i$ with $r_i$.

## 5.1 Optimal DFG covering

Intuitively, optimal DFG covering can be specified as finding a labeling of the variables in $\bigcup_{n_i}(CV_i \cup SR_i)$, w.r.t. minimizing the costs $Cost(DFG) = \sum_{n_i} Cost_i$. It can be shown, that it is sufficient to label the set of variables $V(DFG) = (\bigcup_{n_i} CV_i) \cup \{D_j' | cse(j)\}$, which drastically reduces the search space. Optimal covering can now be specified, making use of ECLiPSe's optimization procedures: $minimize(l(V(DFG)), Cost(DFG))$. The labeling strategy $l(V)$ selects the most constrained variable in each labeling step. We compare three code selection methods for DFGs

| source | CS1 | $t_{CS1}$ | CS2 | $t_{CS2}$ | CS3 | $t_{CS3}$ |
|---|---|---|---|---|---|---|
| exm | 10 | 0.13 | 6 | 0.28 | 5 | 1.33 |
| iir | 16 | 0.4 | 13 | 1.69 | 13 | 1.97 |
| cu | 16 | 0.3 | 12 | 4.80 | 12 | 5.30 |
| cm | 14 | 0.17 | 10 | 6.90 | 10 | 4.32 |
| lf | 39 | 0.7 | 24 | 235.37 | 24 | 687.40 |
| t1 | 44 | 0.69 | 29 | 27107.61 | 28 | 24555.29 |
| t2 | 42 | 0.9 | 28 | *.** | 26 | 53296.18 |
| t3 | 47 | 0.55 | 31 | 1929.09 | 28 | 1490.33 |
| t4 | 148 | 2.65 | 96 | *.** | 88 | *.** |

Table 2: *Results of different code selection methods*

and provide experimental results (w.r.t. a sequential execution model) showing how the proposed technique eliminates the drawbacks of tree-based covering mentioned in section 2. **CS1:** Traditional DFT-based covering, i.e. DFGs are split into DFTs at CSE nodes. Each $DFT_i$ is covered optimally but separately. A constraint $cse(r_i) \Rightarrow D_i' \in Mem$ ensures that CSEs are located to memory. Covering is performed by applying $minimize(l(V(DFT)), Cost(DFT))$. **CS2:** DFG covering, while taking into account the routing costs of CSEs to their uses. As an example, consider fig. 7. The introduction of the extra result location $D'$ of CSEs (cf. section 4.2) leads to further improvements. In fig. 7 the transfer paths from node 0 to nodes 1 and 2 both require a move from SR *ar* to *my*. Both routes are combined in one route from $D$ to $D'$, thus reducing the transfer cost from 2 to 1. Covering is specified by imposing a constraint $cse(r_i) \Rightarrow D_i' \notin VSR$ and applying $minimize(l(V(DFG)), Cost(DFG))$. **CS3:** DFG covering, while additionally allowing CSEs to be mapped to chained operations (fig. 3c). No extra constraints are needed. Table

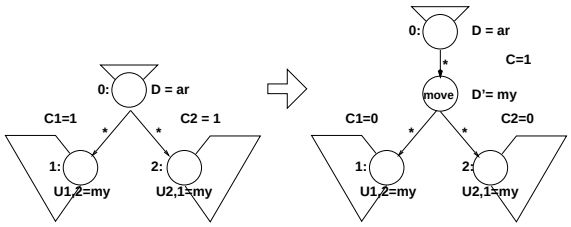


Figure 7: *Effect of CSE routing*

2 shows the resulting costs (number of RTs) and runtimes of the DFG code selection methods CS1 to CS3. The improvements from strategy CS1 to CS3 range within $18\% - 50\%$ (35 % on average). Runtimes shown as "*.**" refer to examples where the optimization did not terminate within 24 hours (thus not proven to be optimal).

## 5.2 Heuristic DFG covering

Since optimal DFG covering is an exponential problem, optimal code selection for DFGs may be too computation-time intensive for large problems. We have therefore designed an additional code selection method, which splits a DFG into a set of smaller manageable DFGs. This method (**CS4**) leads to much better runtimes than optimal DFG covering, while coming close to the optimal results. In CS4, code selection is based on the optimization strategies described in the last subsection. The strategy for partitioning the DFG is splitting the DFG at its CSEs (like in CS1), leading to a set of DFTs. We assume a certain ordering $[DFT_1, \ldots, DFT_n]$ of the DFTs. Now, for each $DFT_i$, the variables $D_i'$ and $TC_i'$ of $r_i$ are excluded from labeling, and variables $D_j'$ and $TC_j'$ of all CSEs $n_j$ being used in $DFT_i$ are included in labeling. An example is shown in fig. 8, where the labeling of $D_1'$ at the root of tree $DFT_1$ is postponed until $DFT_2$ is labeled (note that adding $r_1$ to $DFT_2$ yields a DFG). The results of the method CS4 are shown in table 3, compared to the costs of optimal DFG covering (CS3) and to pure DFT covering (CS1). The costs achieved by CS4 are much better than the costs of CS1 and come very close to the optimal cost values.

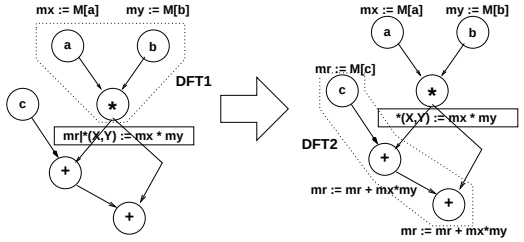


Figure 8: *Covering method CS4*

| source | CS1 | CS3 | CS4 | $t_{CS4}$ |
|---|---|---|---|---|
| exm | 10 | 5 | 5 | 0.30 |
| iir | 16 | 13 | 13 | 0.58 |
| cu | 16 | 12 | 12 | 0.95 |
| cm | 14 | 10 | 10 | 0.46 |
| lf | 39 | 24 | 25 | 4.94 |
| t1 | 44 | 28 | 28 | 2.49 |
| t2 | 42 | 26 | 27 | 2.20 |
| t3 | 47 | 28 | 30 | 3.51 |
| t4 | 148 | 88 | 92 | 216.51 |

Table 3: *Results of CS4*

### 5.3 Phase-coupled code generation

As mentioned in section 2 (fig. 4), selection of only a single optimal DFT or DFG cover from multiple alternative optimal covers may affect exploitation of ILP. A better exploitation of ILP is possible, if the final binding of operations and values to FUs and SRs is postponed until instruction scheduling. This phase coupling can be realized in our approach, since the FRT covering technique introduced in section 4.2 does not commit to a single DFG cover but implicitly retains a set of alternative optimal covers. We have implemented an extended list scheduling algorithm that integrates code selection, register allocation and instruction scheduling. The scheduling algorithm takes a FRT cover and transforms it into a sequence of machine instructions, while adding new constraints (e.g., on ILP) and thus reducing the resource sets. The amount of alternative resources represented by a FRT cover provides a high flexibility for making good decisions in each step during scheduling. We have generated parallel code for the example

| source | GNU | hw | pc | $t_{pc}$ |
|---|---|---|---|---|
| exm | - | 5 | 5 | 0.3 |
| iir | 33 | 12 | 12 | 1.8 |
| cu | 23 | 9 | 9 | 1.9 |
| cm | 16 | 6 | 6 | 0.4 |

Table 4: *Executable code for DSPStone benchmarks*

from figure 8 and some DSPStone benchmarks [16] and the ADSP-210x based on method CS4. Results are given in table 4. Each entry gives a number of generated parallel machine instructions (including additional code for address computations). Column 2 shows results obtained with a GNU-based ADSP-210x C-Compiler. Column 3 (hw) gives the length of the hand-written reference code for the DSPStone benchmarks, while column 4 (pc) provides the results achieved by the phase-coupled code generation technique. Column 5 ($t_{pc}$) shows the runtimes for phase coupled scheduling. For the tested benchmarks our technique was able to produce the same code quality as in the case of the hand-written code.

# 6 Conclusions

The irregular architecture of fixed-point DSPs often prevents compilation of efficient machine code due to many constraints imposed by special-purpose registers and ILP. In order to overcome this problem, in this paper we have proposed a novel constraint-driven approach to code selection, which takes irregularities in a DSP architecture into account. We have shown that the proposed DFG covering technique produces better code (for a sequential execution model) than the traditional tree-based method, due to a more efficient code selection for CSEs. Furthermore, our approach enables phase coupling by exploiting alternative DFG covers during the scheduling phase. Experimental results demonstrate that by exploiting these two features the quality of compiler-generated code can be significantly improved as compared to existing techniques and may come close to the quality of hand-written assembly code.

# References

[1] G. Araujo, S. Malik, and M. Lee. Using Register-Transfer Paths in Code Generation for Heterogeneous Memory-Register Architectures. In *33rd Design Automation Conference (DAC)*. 1996.

[2] A. Fauth, G. Hommel, A. Knoll, and C. Mueller. Global code selection for directed acyclic graphs. In Peter A. Fritzson, editor, *Compiler Construction*, volume 786 of *LNCS*, pages 128–141. Springer–Verlag, Edinburgh, U.K., April 1994. 5'th International Conference, CC'94.

[3] C. Fraser, R. Henry, and T. A. Proebsting. Engineering a Simple, Efficient Code-Generator Generator. *ACM Letters on Programming Languages and Systems*, 1(3):213–226, September 1992.

[4] C.H. Gebotys. An Efficient Model for DSP Code Generation:Performance, Code Size, Estimated Energy. In *10th International Symposium on System Synthesis (ISSS)*. 1997.

[5] S. Hanono, G. Hadjiyiannis, and S. Devadas. Aviv: A Retargetable Code Generator Using ISDL. In *Proc. 34th DAC'97*, 1997.

[6] D. Lanner, M. Cornero, G. Goossens, and H. De Man. Data routing: a paradigm for efficient data–path synthesis and code generation. In *Proc. 7th IEEE/ACM Int. Symp. on High–Level Synthesis*, May 1994.

[7] R. Leupers. *Retargetable Code Generation for Digital Signal Processors*. Kluwer Academic Publishers, 1997.

[8] R. Leupers and P. Marwedel. Retargetable code generation based on structural processor descriptions. In *Design Automation for Embedded Systems, vol. 3, no. 1*, 1998.

[9] S. Liao, S. Devadas, K. Kreuzer, and S. Tjiang. Instruction Selection Using Binate Covering for Code Size Optimization. *International Conference on CAD (ICCAD)*, 1995.

[10] K. Marriott and P.J. Stuckey. *Programming with Constraints: An Introduction*. The MIT Press, 1998.

[11] P. Marwedel and G. Goossens, editors. *Code Generation for Embedded Processors*. Kluwer Academic Publishers, 1995.

[12] P. Paulin, C. Liem, T. May, and S. Sutarwala. Flexware: A Flexible Firmware Developement Envirenment for Embedded Systems. In Marwedel and Goossens [11], chapter 4, pages 65–84.

[13] K. Rimey and P.N. Hilfinger. Lazy Data Routing and Greedy Scheduling. In *MICRO*, volume 21, pages 111–115. 1988.

[14] M. Wallace, S. Novello, and J. Schimpf. ECL$^i$PS$^e$: A Platform for Constraint Logic Programming, 1997. Publications at http://www.icparc.ic.ac.uk/.

[15] T. Wilson, G. Grewal, S. Henshall, and D. Banerji. An ILP-Based Approach to Code Generation. In Marwedel and Goossens [11], chapter 6, pages 103–118.

[16] V. Zivojnovic, J.M. Velarde, C. Schlaeger, and H. Meyr. DSPStone - A DSP oriented Benchmarking Methodology. In *ICSPAT*. 1994.