Global Code Selection for Directed Acyclic Graphs*

Andreas Fauth[1], Günter Hommel[1], Alois Knoll[2], Carsten Müller[1]
fauth@cs.tu-berlin.de

[1] Technische Universität Berlin, Institut für Technische Informatik,
Franklinstr. 28/29, D-10587 Berlin, Germany
[2] Universität Bielefeld, Technische Fakultät,
Postfach 10 01 31, D-33501 Bielefeld, Germany

Abstract. We describe a novel technique for code selection based on data-flow graphs, which arise naturally in the domain of digital signal processing. Code selection is the optimized mapping of abstract operations to partial machine instructions. The presented method performs an important task within the retargetable microcode generator CBC, which was designed to cope with the requirements arising in the context of custom digital signal processor (DSP) programming. The algorithm exploits a graph representation in which control-flow is modeled by scopes.

1 Introduction

In the domain of medium-throughput digital signal processing, micro-programmable processor cores are frequently chosen for system realization. By adding dedicated hardware (accelerator paths), these cores are tailored to the needs of new applications. Optimized processor modules can be reused, which is a major benefit compared to *high-level synthesis* [28] where a completely new design is developed for each application. Because of the application-specific add-ons and the rather short lifetimes of a specific design, there is a need for retargetable software development tools, especially code-generators.

1.1 Overview

In the next section we will shortly discuss several related approaches to code generation and point out some differences of our system. Section 3 introduces the overall architecture and functionality of the CBC code generator. Section 4 explains the code selection task and the basic techniques used. In section 5 our algorithm is presented. We conclude the paper with experimental results.

* Part of this research is supported by the ESPRIT 2260 ("SPRITE") project of the European Community and Siemens AG, München.

2 Related Work

Lansdkov et al. [27] present a machine model and methods for microcode generation. A subtask of code selection called *bundling* and a subset of scheduling called *compaction* are described. Both methods have a local view on the subject program. The YC system [6] deals with code selection but does not provide detailed scheduling. A phase called *combiner* only tries to concatenate adjacent operations. The work of Rimey [31, 32] describes a compiler for application-specific DSPs. The main attention, however, is paid to scheduling and data-routing (i.e. mainly register assignment and spilling). Code selection and scheduling are only performed on straight-line code. Optimizations across branch boundaries are not performed. The MARION system [2, 3] performs code generation for RISC architectures. Here, a simple approach for code selection is chosen. A recursive-descent brute-force tree pattern matcher neither considers graph structure of the intermediate code nor global subexpressions. Our implementation is based on the work of Fraser et al. [13, 14].

Points of major differences between our code selection approach and similar tasks in "classic" code generation (CG) are:

- *Complexity of datapaths.* CBC has to deal with highly specialized and optimized datapaths. The hardware units make the efficient execution of frequently used operation sequences possible. Operation patterns for the functional units of these datapaths are much more complex than for standard microprocessors.
- *Type-handling.* DSP algorithms may employ a large variety of different word lengths and numerical types. The hardware operators are restricted to fixed word lengths. A correct mapping must always be found. In most CG work this topic is neglected because language definitions (and hence the compilers) are restricted to "implementation-dependent" types.
- *Evaluation order.* Approaches like [6, 7] dealing with code selection assume a fixed evaluation order, which is usually derived from the imperative source code. There is no explicit scheduling phase included in the back-ends. Commonly, register allocation is performed during code selection. Most of the time this is done by graph coloring [4] or "on-the-fly".
- *Parallelism of functional blocks.* Most DSP architectures contain several functional units that work in parallel. Therefore, the final code cannot be emitted during or immediately after the code selection phase because *partial* instructions must be "compacted" into *complete* instructions at a later stage of compilation exploiting the possible parallelism. Consequently, code selection must not specify the complete behavior of the machine for each cycle. It must only select code for each of the individual units.
- *Expressions are DAGs.* Intermediate programs formulated in *directed acyclic graphs* (DAGs) pose a problem to classic code selection approaches. "*We assume that the intermediate code and the target code are presented as trees or terms*" [8] is a typical statement. Tree matching methods [1] are popular.

- *Machine description.* In the compiler writer community machine descriptions are mainly intended to be used by the code generator only. Some detailed knowledge of the compiler is necessary to write good descriptions. By giving the semantics for each instruction as a transformation of the machine state, we describe the instruction set in a behavioral way. Out of this machine *description*, various machine *models* can be generated depending on the application (e.g. code generator, assembler or simulator).
- *Intermediate representation.* Our intermediate representation is based on a data- and control-flow graph description that differs from the representations used in many compilers.

3 Anatomy of the Compiler

In CBC, code generation is split into different tasks. Each of these is performed by a specific tool. The intermediate results are passed on in human-readable text files. Figure 1 shows the general layout of the code generator, the underlying data- and rule-base as well as the retargeting mechanism.

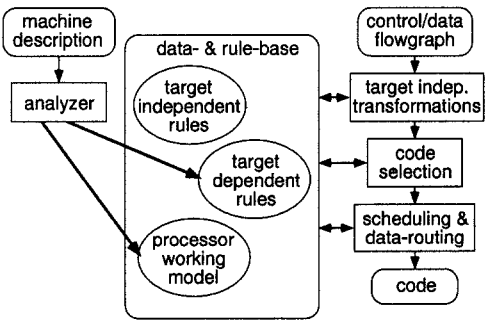

Fig. 1. System overview of CBC's code-generator and its retargeting process.

The primary goal is to generate highly optimized code from the description of the algorithm, which is specified graphically or textually in a signal flow graph. In principle, it is also possible to write the algorithm in other languages that are capable of modeling parallel behavior in an adequate way, e.g. the synchronous subset of the applicative real-time DSP language ALDiSP [16]. The intermediate representation can be easily obtained from a signal flow graph and will be described in section 3.3.

3.1 Retargeting

In our approach, the language `nML` [15] is used to describe the target architecture (see Fig. 2). Originally designed as a simple means for expressing programming

models as found in the usual programmer's manuals, it has turned out to be powerful enough to describe current and future DSP cores – it may even serve as the basis for high-level hardware synthesis [12]. Its main advantage from the programmer's point of view, however, is its compactness combined with its readability. **nML** is intended for describing arbitrary *single instruction stream* architectures. Such architectures feature a single program counter, but can otherwise consist of an "unlimited" number of building blocks. Based on attribute grammars, **nML** is flexible and reasonably easy to use.

```
type word = int(14)
type addr = card(20)

mem REG[8,word]
mem RAM[2**20,word]
mem latch[3,word]

op instruction = jump | aluOp | ...
op alu(a:aluAction,s1:src,s2:src,d:dst)
  action = { latch[0]=s1; latch[1]=s2; a.action;  d=latch[2]; }
  syntax = format("%s %s,%s,%s",a.syntax,s1.syntax,...)
  image = ...

op aluAction = sub | add | ...
op sub()
  action = { latch[2] = latch[0] - latch [1]; }
  ...

mode src = reg | ...
mode reg(n:card(3)) = REG[n]
  syntax = format("R%d",n)
  image = format("01%b",n)
...
```

Fig. 2. Excerpt from an **nML** machine description.

When retargeting the compiler, the **nML** analyzer examines the instruction set and the memory description of the target processor and builds a *machine model*, i.e. a representation of the capabilities and constraints of the machine. The process of building this model is detailed in [10, 12]. The machine model, along with the datapath constraints and machine-independent transformation rules are given as input to the generic (parameterized) code generator. The transformation rules specify, for example, how to perform a 32-bit addition on a 16-bit machine. The phases of code generation and the construction of the generic compiler are outlined in [9, 11].

3.2 Code Generation Script

The main tasks of code generation are:

- *Signal flow graph translation.* This is the algorithmic design entry to the code generator. The specification of the application program is constructed using a schematic editor and a simulation tool. The resultant signal flow graph is translated by this front-end into the code generator's internal data format.
- *Control-flow transformations.* Transformations concerning the mutually exclusive execution of operations depending on certain conditions are performed to reduce the overall execution time. A pure data-driven representation is translated into a hybrid data/control-driven representation reflecting the requirements of branch controllers and conditional transfers used in programmable DSP systems [11, 26].[3]
- *Code selection.* Subsets of the algorithm are mapped to datapaths. First, high-level operations of the algorithmic input are expanded into machine-executable operations. Then, chains of expanded operations are merged to form more complex operations that are provided by the machine. This clustering reduces the complexity of the scheduling task and allows optimized code generation in reasonable time.
- *Scheduling and data-routing.* The operations in the graph are ordered in time. To produce high quality code, efficient scheduling is a necessity. The goal of scheduling is minimum execution time for a given algorithm on an architecture which is fixed at compile-time. Therefore, the assignment of registers to intermediate values, the generation of data-routes (including spill-code) and scheduling are performed in parallel [23, 32].

3.3 Intermediate Representation

The intermediate representation is a *control/data-flow graph* (CDFG). A CDFG is a program description based on a directed graph (N, E) consisting of two finite sets: the nodes $n_i \in N$ represent the operations of the program and the directed edges $e_i \in E$ which are ordered pairs of nodes $e_i = (n_j, n_k)$ display dependencies between the operations. An edge can either model a *data-flow dependency* (i.e. a data flow path) or an additional *control-flow constraint.*[4] The CDFG describes the body of the main execution loop of an application. Cycles in the graph result only from *algorithmic delay operations* which are used to refer to values from earlier incarnations of loops.

The *data-flow graph* models all data dependencies and operations. An operation node can be executed whenever input data is available.[5] Inputs and outputs of the program are represented as data sources and data sinks. Data is

[3] This task is actually split in two: One phase before and one phase after code selection. The first phase rewrites scope structures and the second inserts jump operations.

[4] Note that each data-flow edge implicitly models a control-flow constraint.

[5] All preceding control-flow constraints must also be satisfied.

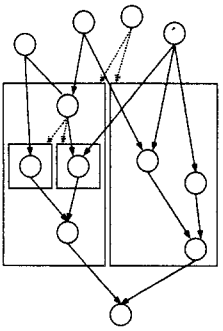

Fig. 3. A CDFG with nested conditional scopes. Each box represents one scope. The two large boxes represent top-level exclusive scopes. The two small boxes represent a pair of exclusive scopes local to the left top-level scope. The nodes represent operations and the arcs represent data-flow edges. The dashed arcs are flag edges from the condition to the true and the false case.

represented as *signals*. A signal represents an *infinite stream* of values. For *synchronous* data-flow, the amount of data produced and consumed for each node is specified a priori. Our data-flow model limits the amout of data produced and consumed in a single cycle to exactly one. The execution of an operation therefore consists mainly of the *use* of one signal at each incoming edge and the *definition* of one signal at each outgoing edge.[6]

The *control-flow graph* is basically a hierarchical structure of *macro nodes*. A macro node is a cluster of operations and other macro nodes. They are used to model loops and conditional *scopes*. All operations inside a specific conditional scope are related to a certain condition. Additionally, *control-flow edges* display precedence relations between operations. At the beginning of code generation there are few control-flow edges; later phases insert additional control-flow information modeling *in-place storage* of signals and the programming of the *branch controller*. The scheduler must find an explicit execution order for all operations, resulting in a sequentially executable microprogram.

For the different stages of code generation, three distinct sets of arithmetic and logic operations exist in a common library:

- *Abstract operations* (AOs). This is set of high-level operations that is available in the initial input-level graph.
- *Machine-executable operations* (MEOs). This set consists of operations which correspond to primitives of the **nML** description. All initial CDFG operations must be mapped to members of this set.
- *Datapath operations* (DOs). The third set comprises operations which occupy a full datapath. They are the basic entities for the scheduling process. These

[6] At this stage of the translation, all multi-rate segments of the program must be translated into loops or unrolled into straight-line code.

operations are formed out of the MEOs during chaining and represent the valid combinations of MEOs.

Besides these operations, some *canonical operations* identifying the action on dedicated hardware (such as accelerator paths) can also be included in the algorithm at each stage of the translation. Since they represent both abstract and datapath operations they are included in the CDFG upon entry to the script and need not be transformed during code selection. Two more groups exist:

- *Transfer operations.* These are used to describe assignments of data to memory locations and moves on buses. They are inserted into the description to route data between different storage locations and correspond to addressing modes and move operations.
- *Control-flow operations.* All conditional and unconditional jumps belong to this set.

4 The Problem of Code Selection

Prior to code selection, the algorithm consists of operations that are machine-independent and well-typed. After code selection, the algorithm must consist of operations that are equivalents for clusters of MEOs. These clusters are associated with datapaths and must not violate encoding restrictions. The first stage of code selection consists of two interleaved phases: *machine-parameterized macro expansion* and *mapping to machine-executable operations.* The second stage maps parts of the algorithm to datapaths.

4.1 The General Approach: Macro Expansion and Chaining

During *macro expansion*, operations in the CDFG are expanded into operations available on the machine. For example, multiplications are broken down to combinations of additions and shifts or into Booth-multiplication steps [24]. This process is controlled by rules, which are parameterized by the set of specific hardware operators offered by the target machine[7]. Therefore, the *rules* are machine-independent, but the *choice* between them is driven by the structure of the target machine.

When *mapping* to MEOs, limited word lengths are taken into account, i.e. the expanded execution of an operation on a smaller word length datapath is constructed. For example, an addition of two 32 bit values could be performed on a 16 bit datapath with two additions (assuming an addition with carry is possible). This task employs the CATHEDRAL-2ND tool for expansion [28]. However, it relies heavily on our own operation library [29], which is two-fold: A machine-independent part describes constant folding and other peephole optimizations; a machine-dependent part describes all MEOs as well as the corresponding expansion rules. The machine-dependent entries are either generated or instantiated

[7] This set is identified during the analysis of the nML machine description.

from templates during the retargeting process. Implementation alternatives are given from which the appropriate expansion can be chosen.

To allow the generation of optimized code within reasonable time, it is important to reduce the complexity of the scheduling task. Therefore, the second part of the code selection task maps subsets of the algorithm onto datapaths prior to scheduling. Once all high-level operations are refined to MEOs, clusters of direct data-dependent operations which can be performed on a datapath within a single cycle are identified. These *chains* of operations are merged and replaced by a single operation each, thus forming more complex operations that are provided by the machine. These *datapath operations* occupy a complete datapath. In Fig. 4 a CDFG is clustered to be executed on the depicted datapath. The shift operations (>>) are executed on the SHIFTER and the arithmetic operations (+ and -) are executed on the ALU CORE.

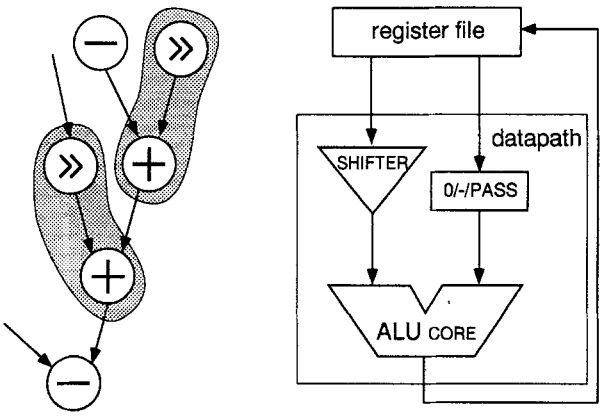

Fig. 4. A CDFG fragment and a datapath

Our chaining process resembles code selection in standard compilers since not all possible combinations of operations are legal chains. Restrictions which must not be violated result from the instruction set definition (see section 4.3).

4.2 Global Chaining

As outlined earlier, the goal of chaining is a "good" assignment of machine operations to datapaths. This implies that chaining *assists* the scheduler; it could indeed be integrated into the scheduling phase at the expense of increased complexity and run time. On the other hand, when chaining is done outside the context of scheduling, little information about resource usage is available. Especially in the presence of multiple *similar* datapaths[8] it is hard to estimate the

[8] Informally, two datapaths are called similar if they share many chaining patterns.

impact of a particular chaining decision on the quality of the resulting code: Operator assignment performed during chaining may result in schedules not fully exploiting potential parallelism of the machine. To decouple the two tasks, the chaining tool must annotate chains with implementation *alternatives*. In this paper we can thus neglect the problem of similar datapaths.

Since the architectures under consideration feature complex datapaths, we emphasize that whole *expressions* are assigned to a single datapath whenever possible. A chaining decision can affect the choices for distant operations, i.e., it has *global* effects. Therefore, large pieces of the CDFG must be considered when making a specific decision.

4.3 Encoding Restrictions

In general, the set of operation tuples executable on a datapath is not equal to all possible combinations of the hardware operators' functionalities; the designer may (and usually will) have imposed restrictions on operation chains. This is quite natural: the number of possible combinations affects the length of the instruction word. It might be necessary to omit some (rarely needed) combinations to reduce the instruction word length. Furthermore, there may be conflicts in the datapath hardware that prohibit certain combinations. As a result, code selection has to comply with encoding restrictions. As it is quite clear that the datapath structure alone is not sufficient to hold this information, we decided to represent legal chains as a set of rewrite rules. Pattern matching is employed to find legal chains in the CDFG.

4.4 Matching on Trees

Pattern matching is an established technique for instruction selection from expression trees in compilers for imperative languages [17, 21]. Code selection for stock microprocessors focuses mostly on a good exploitation of complex addressing modes. In the context of CBC, however, the emphasis is on good utilization of the complex datapaths. Nevertheless, similar tools can be used at the technical level. In the CBC environment, all legal patterns are generated by the `nML` front-end [9] and stored as a set of match-replace pairs (see Fig. 5 for an example). The match-replace database is intentionally held human-readable to allow an experienced user to modify some rules or add new rules by hand (e.g. for special optimizations). The depicted rule does not take commutativity of the `add` operation into account. This is not a serious problem; the `nML` front-end simply generates multiple patterns (in this case, `s = add(t,i2)` is replaced by `s = add(i2,t)`).

In the context of our compiler, the term rewrite system is not one monolithic unit; pattern matching and rewriting are separated phases. The tree parser generator we use, IBURG [14], is only concerned with the matching phase; the connection to the rewrite phase is made by match rule numbers. The tree grammar (from which the tree parser is generated) and the rewrite procedure are both generated by our chaining preprocessor, which takes the rewrite rules as

```
MATCH
  i1 = reg;
  i2 = reg;
  c  = const({-8..7});          /* A value constraint. */
  t  = shift(i1,c);
  s  = add(t,i2);
REPLACE COST=1
  s  = shiftadd(i1,i2,c);
ENDM
```

Fig. 5. A sample rewrite rule for the datapath of Fig. 4.

its input. The incorporated match algorithm is an extension to the BURS (Bottom Up Rewrite System) [30] theory and allows the computation of an optimal rewrite sequence for a tree (by matching the rewrite rules to subtrees), given a fixed set of rewrite rules with fixed costs. This computation takes time linearly proportional to the size of the tree. For the selection of the optimum match, tree parsing with dynamic programming is used [1]. The tree parser generator BURG [13] performs the dynamic programming at parser generation time and thus generates highly efficient pattern matchers. IBURG, a heavily simplified BURG version, still generates very efficient parsers, but their running time is no longer independent of the number, size, and structure of the patterns.[9] Because of its simplicity, IBURG can be modified quite easily. We extended it to accept certain match conditions in the rules; this way we can conveniently express type constraints or other operand constraints which are imposed by the hardware operators.

5 Code Selection on Graphs

Commonly, code selection is performed on expression trees. These are (partial) statements usually directly reflecting source language statements. The programs being compiled in our environment contain a large amount of decision making and common subexpressions. As mentioned above, cycles in the graph only result from values produced by delay operations. These are not considered during code selection.[10] Hence expressions in our CDFG model are DAGs. This means that intermediate results can have more than one use (Fig. 6a) which can also reside in different conditional scopes (Fig. 6b). Figure 6c shows a signal that has multiple definitions in different conditional scopes. In traditional compilers, conditions are at "borders" of basic blocks. The **if-then-else** statements themselves are also subject to code selection. In our approach, operation nodes of the CDFG

[9] However, informally speaking, for our purposes the generated parser have "nearly" linear behavior and are still fast enough.

[10] This is indeed a topic for future investigations.

have a *conditional context*. For each condition a *flag* is computed and connected to a macro node, i.e. a scope. Then, global data-flow is specified, i.e. signals "enter" and "leave" scopes. This representation facilitates code selection that transcends basic blocks.

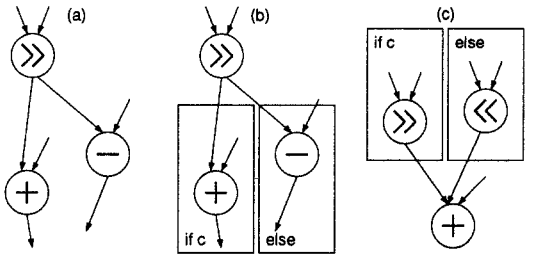

Fig. 6. Cases of interest to our global chaining approach: (a) multiple uses in the same scope, (b) multiple uses in different scopes and (c) multiple definitions.

Consider the architecture depicted in Fig. 4. A value produced by the SHIFTER is not immediately available for more than one operation. For that purpose an addition with zero must be performed to pass the value unchanged through the ALU CORE, i.e. the value is "spilled" to a register. We assume that the datapaths do not fork (and thus do not allow multiple uses within themselves). This implies that an operation defining a multiply used value can never be chained. The CDFG in Fig. 6a would be mapped to three operations (a **shift**, an **add** and a **sub**) instead of two in the optimal case (a **shift-add** and a **shift-sub**). The best results possible for the datapath of Fig. 4 are shown in Fig. 7.

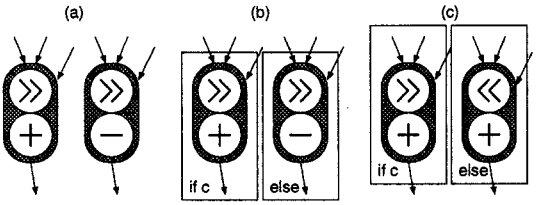

Fig. 7. The best solutions to the three different cases of Fig. 6.

5.1 The Simple Approach: "Undagging"

Earlier section were concerned with *tree* parsing and pattern matching in *trees*. On the other hand, it was unveiled that CDFGs are definitely not tree-like in the general case. The resulting problems have to be solved. Taking the previous section into account, it can be seen that some subgraphs indeed have a tree

structure; namely those that lie between points of multiple uses and multiple definitions. Incidentally, the values which are defined or used more than once must be held in registers: multiple definitions require a proper modeling of control-flow which cannot (generally) be mapped onto the datapath; multiple uses map to different instructions.[11] This leads to a very simple chaining method: Cutting the DAG whenever a value is defined or used in multiple places yields a set of (usually small) trees. These trees are then individually processed by the rewrite system and reconnected afterwards to compose a chained version of the original DAG.

5.2 A More Sophisticated Approach: Heuristic Node Duplication

The advantage of the previous method is its simplicity. However, in a significant number of cases chaining possibilities are lost due to cuts at multiple uses or definitions. We seek a way to improve this situation. The key insight is that the CDFG must be modified in order to create more chaining possibilities.[12] Consider the cases where chaining possibilities may be missed. There are essentially four of them:

- *A signal has one definition and multiple uses in the same scope.* This implies that this signal must be made available to different DOs (since multiple uses in the datapath are not possible). By duplicating the definition once for each use, the multiple use has been resolved (while introducing a multiple use at each operand) and the desired chaining possibilities have been created.
- *A signal has one definition and multiple uses, at least one in another scope.* To generate a chaining possibility, the uses must be within the same scope as the definition. A further look reveals that this case resembles the previous one; it is solved in basically the same manner.
- *A signal has multiple definitions (in mutually exclusive scopes) and one use outside the scopes of the definitions.* A chaining possibility can be created by duplicating the use and nesting the copies into the scopes of the definitions. However, we must bear in mind that if the use has yet another operand multiply defined in different scopes, a particular evaluation order for the mentioned defining scopes would be enforced. This could be undesirable (see Fig. 8a).
- *A signal has multiple definitions (in mutually exclusive scopes) and multiple uses.* This case is not further considered since there are rarely any cases where duplication of operations could lead to shorter code. Consider a signal that has n uses and m definitions. If each of these operations is (trivially) chained to a single DO, we get $n + m$ operations. If all necessary copies of operations are generated to get more chaining possibilities and each of these would actually be chained, the result would be $n \times m$ DOs (see Fig. 8b).

[11] This is a consequence of the postulation that no multiple uses exist in the datapaths.

[12] More exactly, we do not want to create chaining possibilities *per se* but only in those places where this will lead to an improvement of the generated code.

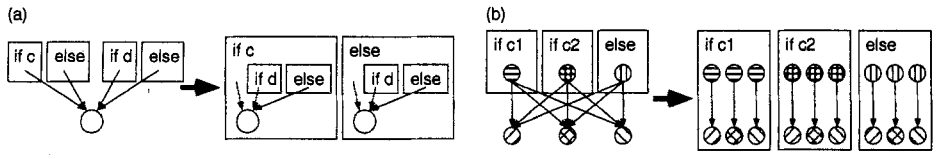

Fig. 8. Node duplication examples. Conditions c and d in case (a) are independent of each other. Conditions c1 and c2 in case (b) are exclusive to each other.

It can be seen that the creation of chaining possibilities is associated with duplication of nodes. When duplicating excessively, the graph might grow too large. This is overcome by first partitioning the graph, which yields (usually) small partitions, and then processing each partition in turn. The partitions are chosen so that no chains across partition boundaries are possible.

One problem not yet mentioned is the *identification* of chaining possibilities. A simple heuristic is employed: For all pairs of MEOs, the pattern base is looked up counting the occurrences of an edge between both operations. (This is quite informal, but should be intuitively justified.) This information is then used for partitioning; two operations are put into the same partition if a chaining possibility exists. Partitions including only one operation are *trivial cases.*

When duplicating into scopes (either at a multiple definition or at a multiple use), the code size might be increased but (usually) not the execution time because only one of the exclusive scopes is executed. To the case shown in Fig. 6a, however, this argument cannot be applied. Therefore, a *common subexpression elimination* (CSE) phase, which succeeds the pattern matcher, removes most of the unchained operation copies from the graph. This also works for duplications at scope boundaries. Since with duplication there is a danger that the number of operations (and thus program size) increases unduly, a cost function is used to resort to the simple undagging method in cases where the node duplication heuristic fails. The cost is computed for both the undagging and the node duplication method[13] as the weighted sum of the number of resulting DOs (a rough estimate of the code size) and the expected number of executed DOs on each execution path (a rough estimate of execution time). The better alternative is kept.

6 Results

The experimental results shown in Table 1 are taken from a "real-life" ADPCM algorithm, which is incorporated into speech compression applications. The (exemplary) datapath from Fig. 4 served as target architecture. The tool was implemented and tested on a SPARC station IPX using C++. All CPU times including parsing and computation of statistics are less than one minute. Therefore, they are not explicitely given.

[13] This is not overly expensive, as the pattern matching and rewriting is quite fast.

Table 1. Experimental results

design	machine-ex. opns.	datapath operations	
		undagging	node dupl.
ENCODE	514	417 (-18.9%)	399 (-22.4%)
ADQUANT	55	51 (-7.3%)	51 (-7.3%)
IADQUANT	21	16 (-23.8%)	15 (-28.6%)
PREDICT	317	262 (-17.4%)	249 (-21.5%)
TONE_DET	25	17 (-32%)	17 (-32%)

7 Conclusion

We have presented an algorithm for code selection on control/data-flow graphs. The approach is based on a global view on the subject programs. The points of interest are multiple uses of values resulting from common subexpressions and multiple definitions of values resulting from conditional scopes. An implementation of the algorithm is incorporated into the CBC compiler and was successfully tested with the Siemens DECT (Digital European Cordless Telephone) design. One line of future research includes the coupling of code selection and scheduling as well as the adaption of our technique to loops.

References

1. A.V. Aho, M. Ganapathi, S.W. Tjiang: Code Generation Using Tree Matching and Dynamic Programming. ACM TOPLAS **11**:4 (1989) 491–516
2. D.G. Bradlee, R.R. Henry, S.J. Eggers: The Marion System for Retargetable Instruction Scheduling. Proc. PDLI'91, SIGPLAN Notices **26**:6 (1991) 229–240
3. D.G. Bradlee, S.J. Eggers, R.R. Henry: Integrating Register Allocation and Instruction Scheduling for RISCs. 4th Int. Conf. on Arch. Support for Prog. Lang. and Operating Systems (1991) 122–131
4. P. Briggs: Register Allocation via Graph Coloring. Ph. D. Thesis, Rice Univ., Houston, Texas (1992)
5. R.G.G. Cattell: Automatic Derivation of Code Generators from Machine Descriptions. ACM TOPLAS **2**:2 (1980) 173–190
6. J.W. Davidson, C.W. Fraser: Code Selection through Object Code Optimization. ACM TOPLAS **6**:4 (1984) 505-526
7. H. Emmelmann, F.-W. Schöer, R. Landwehr: BEG – a Generator for Efficient Back Ends. Proc. PLDI'89, SIGPLAN Notices **24**:7 (1989) 227-237
8. H. Emmelmann: Code Selection by Regularly Controlled Term Rewriting. Code Generation – Concepts, Tools Techniques, Springer (1992) 3–29
9. A. Fauth, A. Knoll: Automated Generation of DSP Program Development Tools Utilizing a Machine Description Formalism. Technical Report 1992/31, Technische Universität Berlin, Fachbereich 20, Informatik, Berlin (1992)
10. A. Fauth, A. Knoll: Automatic Generation of DSP Program Development Tools Using a Machine Description Formalism. Proc. IEEE Int. Conf. on Acoustics, Speech and Signal Processing (1993) 457–460

11. A. Fauth, A. Knoll: Translating Signal Flowcharts into Microcode for Custom Digital Signal Processors. Proc. Int. Conf. on Signal Processing (1993) 65–72
12. A. Fauth, M. Freericks, A. Knoll: Generation of Hardware Machine Models from Instruction Set Descriptions. VLSI Signal Processing VI (1993) 242–250
13. C.W. Fraser, R.R. Henry, T.A. Proebsting: BURG – Fast Optimal Instruction Selection and Tree Parsing. ACM SIGPLAN Notices **27**:4 (1992) 68–76
14. C.W. Fraser, D.R. Hanson, T.A. Proebsting: Engineering a Simple, Efficient Code Generator Generator. ACM Letters on Prog. Lang. and Systems **1**:3 (1993) 213–226
15. M. Freericks: The nML Machine Description Formalism. Technical Report 1991/15, Technische Universität Berlin, Fachbereich 20, Informatik, Berlin (1991)
16. M. Freericks, A. Knoll: Formally Correct Translation of DSP Algorithms Specified in an Asynchronous Applicative Language. Proc. Int. Conf. on Acoustics, Speech and Signal Processing (1993) 417–420
17. M. Ganapathi, C.N. Fischer: Description-driven code generation using attribute grammars. Proc. of the 9th POPL (1982) 108–119
18. M. Ganapathi, C.N. Fischer, J.L. Hennessy: Retargetable Compiler Code Generation. Computing Surveys **14**:4 (1982) 573–592
19. M. Ganapathi, C.N. Fischer: Affix Grammar Driven Code Generation. ACM TOPLAS **7**:4 (1985) 560–599
20. M. Ganapathi, C.N. Fischer: Integrating Code Generation and Peephole Optimization. Acta Informatica **25** (1988) 85–109
21. R. Giegerich: Code selection by inversion of order-sorted derivors. Theoretical Computer Science **73** (1990) 177–211
22. R.S. Glanville, S.L. Graham: A new method for compiler code generation (Extended Abstract). Conf. Record of the 5th POPL (1978) 231–240
23. R. Hartmann: Combined scheduling and data routing for programmable ASIC systems. Proc. European Design Automation Conference EDAC'92 (1992)
24. J.L. Hennessy, D.A. Patterson: Computer architecture: a quantitative approach. Morgan Kaufmann Publishers (1990)
25. C.M. Hoffmann, M.J O'Donnell: Pattern Matching in Trees. JACM **29**:1 (1982) 68–95
26. M. Rim, R. Jain: Representing Conditional Branches for High-Level Synthesis Applications. Proc. 29 Design Automation Conf. (1992) 106–111
27. D. Landskov, S. Davidson, B.D. Shriver, P.W. Mallet: Local microcode compaction techniques. Computing Surveys **12**:9 (1980) 261–294
28. D. Lanneer, F. Catthoor, G. Goossens, M. Pauwels, J. Van Meerbergen, H. De Man: Open-ended System for High-Level Synthesis of Flexible Signal Processors. Proc. European Design Automation Conf. EDAC'90 (1990) 272–276
29. G. Meyer-Berg: The Library LIB for the Hardware Independent Compiler CBC. ESPRIT-II Project 2260 "SPRITE" Report CBC.b/Siemens/Y4m12/2 (1992)
30. T.A. Proebsting: Simple and efficient BURS table generation. Proc. PLDI'92, SIGPLAN Notices **27**:6 (1992) 331–340
31. K. Rimey, P.N. Hilfinger: A Compiler for Application-Specific Signal Processors. VLSI Signal Processing III (1988) 341–351
32. K. Rimey, P.N. Hilfinger: Lazy data routing and greedy scheduling. 21st Annual Workshop on Microprogramming MICRO-21 (1988) 111–115
33. Discussion: Code Generator Specification Techniques. Led by Chris Fraser, Summarized by J. Boyland and H. Emmelmann, Code Generation – Concepts, Tools Techniques, Springer (1992) 66–69