TRANSLATING SIGNAL FLOWCHARTS INTO MICROCODE FOR CUSTOM DIGITAL SIGNAL PROCESSORS†

A. Fauth and A. Knoll
Technische Universität Berlin, Sekr. FR 2-2
Franklinstr. 28, W–1000 Berlin 10, Germany

ABSTRACT

The retargetable microcode compiler CBC suited for application specific DSP (ASDSP) system software development is presented. The compiler is capable of generating code for various single instruction architectures. During the retargeting of the compiler a formal description of the target machine is analyzed in order to extract the machine-specific information needed for code generation. The tasks of the compiler comprise control flow transformations, code selection, data-routing and scheduling, address assignment and code mapping. The problems are discussed, the structure of the microcode compiler is given and future research directions are briefly outlined.

INTRODUCTION

In the domain of medium-throughput DSP (e.g. telecommunications, consumer electronics), microprogrammable processor kernels with additional dedicated hardware units are a good choice for system realization. Advanced reusable processor kernels are parameterized in their number of processor building blocks (e.g. ALUs, AGUs), in the instruction set, in their bus structure and their bus width. Obviously, every change in the kernel architecture entails the need to adapt and hence completely rewrite the software tools used to program it (assembler, debugger, simulator, compiler). The approach taken here to solve the problem of adapting the tools is based on a machine description, which contains all machine-specific information necessary for efficient code generation. Our description formalism nML [1] was designed with the goal of compactness and readability in mind. The formalism is powerful enough to enable compilers being generated automatically solely out of this description [2][3]. Every change of the machine requires only a change of the corresponding part of the machine description.

Once a compiler was generated, code can be produced. The code generation script consists of several tools; each performing a specific task. The user can thus iterate the script and refine the algorithm in parallel with the architecture. The tools provide the programmer with statistics. These can be displayed in various diagrams in order to give a rapid overview about an implementation decision's impact on the design (e.g. overall execution time and hardware operator loads).

MACHINE DESCRIPTION AND ITS ANALYSIS

The target machine is described in terms of its instruction set; a description is thus related very directly to the standard description as found in the usual programmer's manuals. To allow for easy retargetability, the target machine is concisely described by its instruction set. The formalism nML was designed for the modeling of single instruction stream architectures. nML is based on attribute grammars allowing for sharing of common properties (e.g. addressing modes) between different instructions. Fig. 1 gives an example.

The description of the target processor is analyzed to extract all information necessary for code generation [2][3]. As some stages of the compiler are based on information deduced in this step, this task comprises the retargeting of the code generator. Mainly the following information is gathered:

- A set of **hardware entities** representing the operators and the interconnection netlist.
- A set of **operations** is created for each hardware operator and for each complete datapath. These sets are merged forming the base of code selection.
- A set of **match-replace patterns** displaying chains of operations that can be executed on a specific datapath within a single cycle.

```
op instruction = jump | aluOp | ...
op aluOp (a:aluAction,s1:src,s2:src,d:dst)
  action = { tmp1 = s1; tmp2 = s2;
             a.aluAction; d = tmp3; }
  syntax = format("%s %s,…", a.syntax;...)
  image  = format("...
op aluAction = plus | minus | ...
op plus ()
  action = { tmp3 = tmp1 + tmp2; }
  syntax = "add"
  image  = "00000"
mode src = reg | ...
mode reg (n:card(3)) = R[n]
  syntax = format ("D%d",n)
  image  = format ("%3b",n)
```

Fig. 1: A fragment of an nML instruction set description.

CODE GENERATION CONSTRAINTS

The retargetable microcode generator is part of a design environment for application specific digital signal processors. The design philosophy is based on predefined but tunable, i.e. parametric, instruction set processor kernels. Therefore, several constraints are imposed on the code generation tasks:

- **Encoding restrictions**. With respect to the size of the microprogram ROM the length of the instruction word is crucial. Vertical encoding and the removal of several combinations of micro-operations from the instruction set are accomplished to shorten the instruction word. Conflicts in the use of instruction word fields can then arise.
- **Limited register resources and fixed interconnections**. The size of register files and the hardware operator

† Part of this research is supported by the ESPRIT 2260 ("SPRITE") project of the European Community.

netlist are fixed at compile-time. The resulting restrictions on the placement of temporary values must not be violated. The set of data routes is restricted and some registers may not be spillable. Deadlock situations must be avoided and conflict-free data routes must be found during scheduling.

OVERVIEW OF THE COMPILER

The intermediate representation of the algorithm common to all code generation tasks is based on a *control/data flow graph* (CDFG). A CDFG is a directed graph with nodes representing operations and arcs displaying dependencies between nodes. These dependencies are related either to control or to data flow. The CDFG describes the main execution loop of an application. Therefore, cycles in the graph result from *algorithmic delay operations* which are used to refer to values from earlier incarnations of loops. The framework described in [4] is used as an implementation backbone.

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

- **Abstract operations**. This set of operations mirrors the high-level operations which are available in the initial input-level flowcharts. Besides these machine-independent operations, some *canonical operations* identifying the action on dedicated hardware (such as accelerator paths) can also be included in the algorithm.
- **Machine executable operations**. 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**. The third set are operations which use all operators on a datapath. They are the basic entities for the scheduling process. These operations are formed out of the second set during chaining.

Besides the different sets of arithmetic and logic operations, 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 CDFG during data routing.
- **Control flow operations**. All conditional and unconditional jumps are members of this set.

Next, the main tasks of code generation are given (there exists a special tool for performing each of the operations mentioned):

- **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 used in programmable ASDSP systems.
- **Code selection**. Subsets of the algorithm are mapped to datapaths. First, high level operations of the algorithmic input are expanded into primitive machine operations. Then, chains of expanded operations are merged forming more complex operations provided by the machine.
- **Scheduling**. The operations in the signal flowchart are ordered in time. To produce high quality code, efficient scheduling is a must. The goal of scheduling is minimum execution time for a given algorithm for an architecture which is fixed at compile-time [5].
- **Code mapping**. At the last stage of code generation, the code is mapped to program memory and a specific encoding (binary or assembly code) is emitted.

Besides these, an additional tool exists for **user interaction**. A good user-feedback is essential. Diagrams showing operation quantities, operator loads and other details of the algorithm are generated automatically as different stages of translation are passed. As some tools can be parameterized, an environment for convenient control of their functionality is provided.

We will now look at the main tasks of code generation in greater detail.

1. Control flow transformations

An optimal schedule keeps all processor building blocks working in parallel as often as possible. To enable the scheduler to order instructions so as to achieve this goal, the data flow graph is modified and enriched with hints to the scheduler. A typical simple example for optimization arises from the situation shown in Fig. 2.b: two operands `o1` and `o2` are to be either added or subtracted depending upon the value of a condition `c`. For an arbitrary reason, the value of `c`, however, can be determined only after both the subtraction and the addition could be carried out. Furthermore, two adders are available to perform the operation at this particular instant in time. Then it makes sense to actually carry out both operations and take the result depending on `c` after its value is known rather than to wait for `c` to be determined and only then perform the subtraction or the addition.

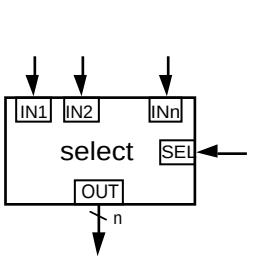

Fig. 2.a: The template for an n-way select operation.

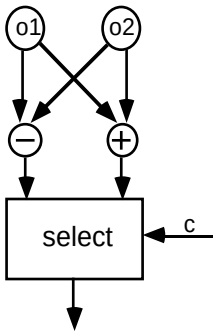

Fig.2.b: An example for conditional selection of a computed value.

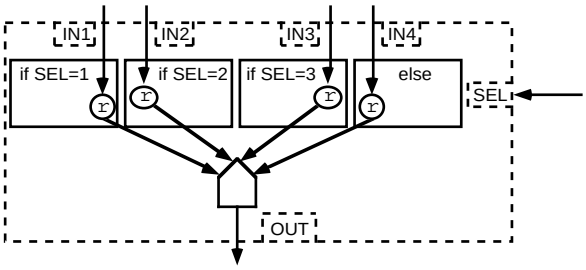

Fig. 2.c: Four different definitions of an operand are merged. The definitions are merely renamings. Each operand depends upon a condition which is the equality of the selector with the appropriate setting.

This example serves only to clarify the issue of reordering conditional computations to obtain better schedules. For the CBC environment a technique for data-dependent clustering and conditional branch handling has been developed. It can be considered as a generalization of conventional instruction reordering techniques like delayed branching, jump look-ahead and multiple prefetching, which are all confined to a rather small scope in the vicinity of branches. In the initial flowcharts, conditional computations are expressed via `select` operations (see Fig. 2.a). The input to the selector SEL determines which of the operands is transferred to

the output. Fig 2.c shows the atomic interpretation of a 4-way select.

Although the `select` operation is convenient for the definition of algorithms, its semantics are not strict: It is obviously not necessary to carry out all operations along all (independent) paths leading to the `select` node. It is therefore desirable *both* to defer the operations along alternative paths until the decision can be made which path is to be chosen *and* to pursue the paths as far as possible before the decision is taken if and only if processor resources are available. A reasonable compromise must be found to ensure an optimized schedule.

To achieve this, for each selectable input operand the set of all operations required to generate the operand is clustered and marked with the appropriate condition by introducing *scopes*. All operations depending on the same condition are within the same scope. This scope is indicated by a surrounding box. The different definitions of an operand are merged in the enclosing scope. Figures 3.a and 3.b show an example where a hierarchy of conditions is established.

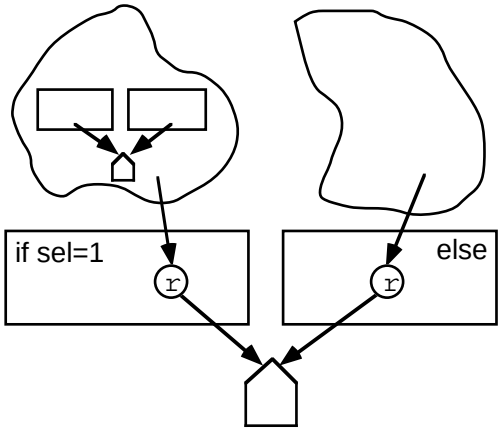

Fig. 3.a: Conditional operations and data-dependent clusters prior to the transformation

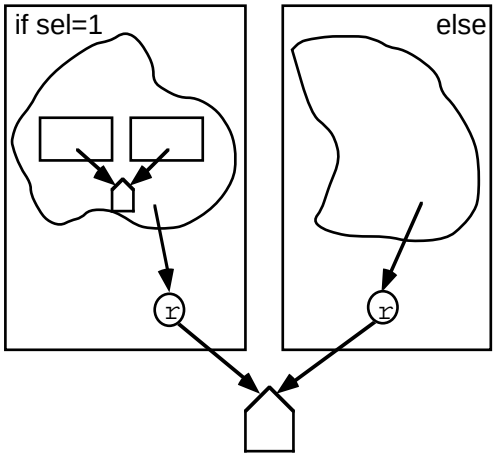

Fig. 3.b: The clusters are marked with the conditions. This transformation is applied recursively to the CDFG possibly resulting in nested conditions.

A pure control-driven interpretation of these clusters would force all operations inside a scope to be delayed until the decision can be taken. This would again result in a sub-optimal schedule. (Similar considerations are given in [6].) Our model for scheduling is based on a hybrid data/control-driven `select` operation: Binary conditional jumps are inserted into the CDFG. This concept can be adapted to n-ary jumps. Two arcs (in Fig. 4 the arcs ending in black dots) emanate from a conditional jump node. One is labeled TRUE, the other is labeled FALSE. These arcs point to operation nodes meaning that the execution of a node is only *necessary* when a certain control flow is selected but can *possibly* be scheduled if it is ready even though the condition has not yet been computed. These arcs therefore do *not* represent true dependencies. They are used to identify mutually exclusive operations and to steer the priority functions of the list-scheduler.

An unconditional jump is also inserted for each pair of mutually exclusive control flows implementing the back jump to the enclosing scope. Control flow arcs (the dashed arcs in Fig. 4) are inserted from all operations of exclusive scopes to the back jump, forcing it to be scheduled after these operations. This enforces the simultaneous completion of all exclusive scopes. The independence of execution times from selected execution paths is ensured which is important to guarantee a conflict-free schedule.

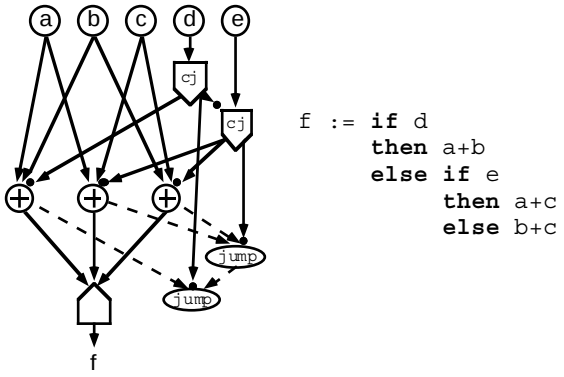

Fig. 4: A CDFG with jumps inserted and the corresponding nested conditional expression.

2. Code Selection

Code selection consists of two interleaved phases: *machine parameterized macro expansion* and *mapping to machine executable operations.*

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. This process is controlled by rules which are parameterized by the set of specific hardware operators offered by the target machine. This set is identified during the analysis of the nML machine description.

In the *mapping* to machine executable operations, limited word lengths have are taken into account. 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 is based on the framework presented in [4].

Once all CDFG operations are refined to machine executable operations, the mapping of operation clusters to datapaths takes place. Thereby, chains of operations are merged, thus forming more complex operations which are provided by the machine. These *datapath operations* can be executed within a single clock cycle. In Fig. 5 the 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.

The basic technique is based on tree parsing and dynamic programming [7]. We enhanced the BURG tree parser generator to generate code including match conditions [8].

These conditions are used to catch *structural properties* and *operator constraints*. Graph patterns can be matched to graphs and type constraints can be expressed. The database which forms the base of this task is a set of match-replace pairs (see Fig. 6.a and 6.b). Two possibilities for graph patterns can be identified:

- In "pure" chaining all operators of a specific datapath are involved in the matched operation. Here, graph patterns exits only if the datapath is graph-like.
- Local optimizations are included in the patterns. This is the case when some local transformations make only sense in combination with a specific mapping (Fig. 6.b).

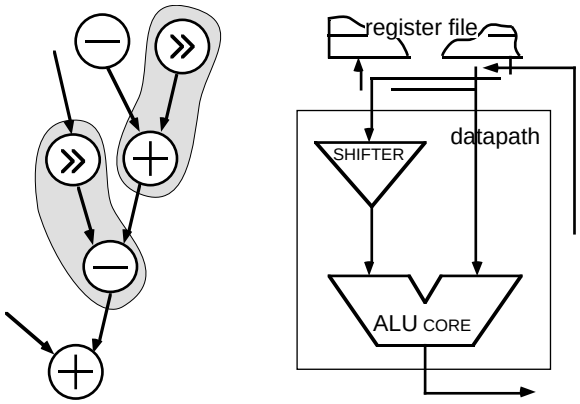

Fig. 5: A CDFG with possible clustering and a simplified datapath for which the clustering was constructed.

```
MATCH
  c const val={-4..3};
  s shift ops=(i1 c);
  n neg   ops=(i2);
  o add   ops=(s n);
REPLACE
  o shsub ops=(i1 c i2);
END
```

Fig. 6a: A match-replace pair with a constrained shift range.

```
MATCH
  a add ops=(i1 i1);
  o add ops=(a i2);
REPLACE
  c const val=1;
  o shadd ops=(i1 c i2);
END
```

Fig. 6b A match-replace pair with a graph pattern. Here, a common sub-expression is resolved.

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

- **Encoding restrictions**. Not all possible combinations of operations on the hardware operators in a certain datapath are allowed whereas in HLS the instruction set is synthesized by the compiler.
- **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 can possible. Operation patterns for these datapaths are much more complex than for standard micro processors..
- **Bit-true mapping**. DSP algorithms may include a large variety of different word lengths. The operators are restricted to processable word lengths. A correct mapping must always be found. In HLS the appropriate hardware is allocated. In CG this topic is of no interest because most compilers are restricted to "implementation-dependent types".

3. Scheduling and data-routing

In our compiler, scheduling is the ordering of datapath operations. The modified list scheduling scheme reported in [5] is adapted to the aforementioned handling of conditions. Data-routing is the combined task of register allocation and generation of *transfer operations*. For load/store architectures, all datapath actions read from and write to registers. Here, the proper transfer operation to move values between RAM/ROM and registers must be inserted into the program. For ASDSP this task can be even more complicated as *critical resources* may exist in the design. Incorrect use of these resources may result in deadlocks of the schedule. Therefore, scheduling and data-routing are performed in parallel.

4. Address assignment and code mapping

Finally, symbolic values are mapped onto physical addresses. RAM usage is optimized by performing a lifetime analysis for all values. Code is mapped to program memory, jump distances are computed and the appropriate encoding is emitted. The encoding functions make use of the object oriented approach. The `syntax` and `image` attributes of the machine description were transformed into methods belonging to the according datapath operations. Now, all operations in the CDFG can simply be "dumped" to emit the proper code.

CONCLUSION

We have shown how microcode generation is performed in CBC. Mappings and optimizations are based on machine independent techniques, parameterized tools and generated methods. The use of a formal machine description allows for easy retargetability giving the system designer good possibilities to experiment with the design. Future efforts will concentrate on the refinement of the presented techniques. Code selection and scheduling are of special interest as is the inclusion of multiple datapath and Harvard architectures.

[1] M. Freericks, "The nML Machine Description Formalism", Tech. Rep. 1991/15, TU Berlin, Germany

[2] A. Fauth, A. Knoll, "Automated generation of DSP program development tools using a machine description formalism", in Proc. ICASSP 1993

[3] A. Fauth, A. Knoll, "Automated Generation of DSP Program Development Tools Utilizing a Machine Description Formalism", Tech. Rep. 1992-31, TU Berlin, Germany

[4] D. Lanneer et al., "An Object-Oriented Framework supporting the full High-Level Synthesis Trajectory", in Proc. CHDL 1991

[5] R. Hartmann, "Combined scheduling and data routing for programmable ASIC systems", in Proc. EDAC 1992

[6] M. Rim, R. Jain, "Representing Conditional Branches for High-Level Synthesis Applications", in Proceedings 29th DAC, 1992

[7] C.W. Fraser, R.R. Henry, T.A. Proebsting, "BURG – Fast Optimal Instruction Selection and Tree Parsing", ACM SIGPLAN Notices, Vol. 27, No. 4, 1992

[8] C. Müller, A. Fauth, "Pattern Matching Based Chaining in Dataflow Graphs", ESPRIT 2260 Tech. Rep. CBC.b/Siemens/Y4m12/1, 1992