# OPTIMIZING COMPLEX MACHINE INSTRUCTIONS WITH DYNAMIC TRELLIS DIAGRAMS

*Stefan Fröhlich and Bernhard Wess*

Institut für Nachrichtentechnik und Hochfrequenztechnik
Vienna University of Technology
Gußhausstraße 25/389, A-1040 Vienna, AUSTRIA

Email: Stefan.Froehlich@nt.tuwien.ac.at

## ABSTRACT

In this paper, we present the application of dynamic trellis diagrams (DTDs) to automatic translation of data flow graphs (DFGs) into highly optimized programs for digital signal processors (DSPs). The concept of DTDs, which is superior to static trellis diagrams used in previous algorithms, is extended to support complex machine instructions such as the MAC operation.

In contrast to static trellis diagrams (STDs), which may be precalculated, DTDs are built at run-time and adapted exactly to the local requirements. This allows to support arbitrary complex machine instructions consisting of several atomic operations which is not possible with STDs. As a result, DTDs are more flexible. Additionally, they need less program memory than STDs. Due to the significant reduction in memory size, the (unavoidable) increase of compilation time is only moderate. At present, the concept of DTDs has been successfully applied to DFG compiler implementations for a variety of general purpose DSP families, including Motorola's DSP56000 and Analog Devices' ADSP2100.

## 1. INTRODUCTION

Programs for DSPs are frequently written in assembly language, even though compilers for high level languages do exist. The reason is that most compilers are not performing well with respect to code size (and therefore run-time). A typical handcoded program will run at about a magnitude faster if coded by an experienced assembly-level programmer. The reason for this behaviour is due to the special architecture of DSPs which makes the implementation of compilers based on standard compiler techniques rather difficult. Additionally, algorithms which run on DSPs are rather time critical, so there is a need to exploit every single instruction cycle which can be saved.

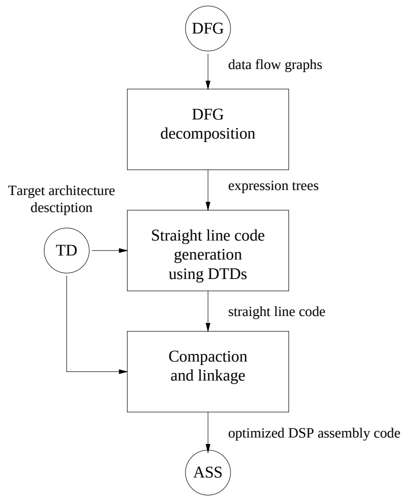
DFG
data flow graphs
DFG decomposition
expression trees
Target architecture desctiption
TD
Straight line code generation using DTDs
straight line code
Compaction and linkage
optimized DSP assembly code
ASS

Figure 1: Basic blocks of the DFG compiler

Programming by hand delivers high quality results, but it is time consuming and error prone, so new solutions to achieve highly-optimized code are desirable.

To produce better results, a DFG compiler has been developed which is able to produce optimum assembly code for a single expression tree and highly-optimized assembly code from a complete DFG specification. An overview of the compiler is given in Figure 1. It consists of a DFG decomposition unit which splits the DFG into one or more expression trees (ETRs) [1, 2], the straight-line code generation which includes the dynamic trellis diagrams discussed in this paper and a postprocessing step to perform code compaction and the linkage of the code fragments for each expression tree.

The tree decomposition has to be done because the generation of optimal assembly code directly from DFGs has shown to be NP-hard [3]. Figure 2 gives a

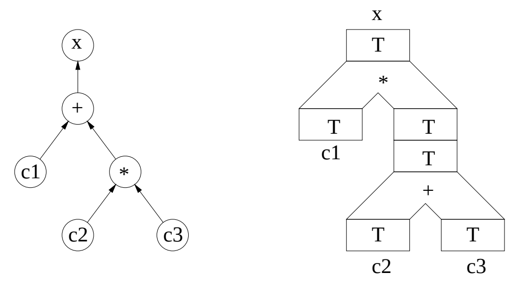

Figure 2: Exptression tree and trellis tree for $x = c1 + c2 * c3$

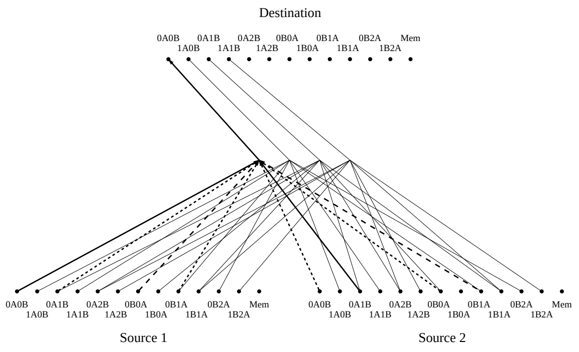
Destination
0A0B 1A0B 0A1B 1A1B 0A2B 1A2B 0B0A 1B0A 0B1A 1B1A 0B2A 1B2A Mem
0A0B 1A0B 0A1B 1A1B 0A2B 1A2B 0B0A 1B0A 0B1A 1B1A 0B2A 1B2A Mem
Source 1
0A0B 1A0B 0A1B 1A1B 0A2B 1A2B 0B0A 1B0A 0B1A 1B1A 0B2A 1B2A Mem
Source 2

Figure 3: STD of a simple add instruction

very simple example for an expression tree and the corresponding trellis tree, consisting of an addition and a multiplication, forming a typical MAC instruction.

After the tree decomposition is finished and all expression trees are identified, locally optimal code is generated for each of the trees using the proposed algorithm, which has an $O(n)$ run-time behaviour [4, 5], $n$ denoting the number of nodes in the DFG. At the beginning of this step, the trellis tree (which is shown on the right side of Figure 2) is generated by concatenating the appropriate trellis diagrams and inserting transfer diagrams in between them. As the generated code is optimal for each trellis tree, it is useful to select the trees as large as possible in order to achieve to best global performance.

As a last step, the locally optimal code segments are concatenated to match the original DFG. Afterwards a compaction algorithm combines two or more instructions into one large instruction word whenever possible, thus generating highly-optimized code for the DFG.

The DFG compiler has been made machine independent by using a behavioural target architecture description. This description is specified in a special language (TDL, [6]), designed to fit the requirements of the compiler. Retargeting can be performed by simply replacing the target description. At present, target descriptions exist for a subset of the functionality of Motorola's DSP56000, Analog Devices' ADSP2100, Texas Instruments' TMS320C5x and NEC's 7701.

## 2. RELATED WORK AND BASIC OPERATION

The first DFG compilers based on the trellis algorithm have been using static trellis diagrams. Trellis diagrams are used as building blocks for the trellis trees which are the basic data structure in the code generation process, containing all the necessary information about the target architecture. An example of a static trellis diagram is shown in Figure 3, describing the add operation of a simple DSP architecture with four registers $A_0, A_1, B_0$ and $B_1$, which are organized in two banks $A$ and $B$. The registers within a bank are symmetric, which means that they are completely interchangable. The trellis diagrams consist of nodes and edges. Each node (which is also called state) corresponds to a certain configuration of allocated and available registers. Additionally it defines the register used by the current result. The edges of a trellis diagram correspond to an instance of an arithmetic or logic operation, connecting two or more processor states.

The operation shown in Figure 3 corresponds to the assembly instruction $A = A + B$. At the top of the figure the destination states can be found. Within the scope of this example, only the bold lines leading to destination state *0A0B* are considerd, indicating that none of the registers are locked and the result is stored into register *A*. In the notation used here, the register which is written first also contains the current operand. As the registers $A_0$ and $A_1$ are symmetric there is no need to decide which register to use at this point of time, as long as we can guarantee that there is at least one register of set *A* available. The numbers in the state description indicate the number of registers *not* available. So *0A* means that no register of register bank *A* is used yet, thus two registers are free to be used.

The two lines at the bottom of Figure 3 represent that machine states for the source operands of the current operation. The notation used is the same as for the destination operand. It is essential that the connection between source and target states reflect the correct processor states. If as an example the left operand is evaluated first, all registers, which are available after the operation has completed, are available while calculating this operand, too. For the evaluation of the right operand, which is done afterwards, there is one less register available because it has to keep the value for the first operand. Figure 3 reflects this by using state *0A0B* for the first source as well as for the destination (indicating that the operand is stored in a register of bank *A*) and by using state *0B1A* for the second operand. The latter state indicates that one register of bank *A* is locked as well as that the second operand is stored in a register of bank *B*.

The pair of dashed lines shows another alternative to evaluate the given expression, calculating the second operand first and the first operand afterwards, using the appropriate states. Furthermore, one could use two more combinations by rewriting the formula as $A = B + A$, which actually swaps the operands. These combinations are shown in dotted style.

If not all registers are available for the computation of the destination, a different state must be used as a starting point for the trellis algorithm. The procedure shown above has to be repeated for all possible destination states to receive the complete trellis diagram as shown in Figure 3.

The number of states within a trellis diagram largely depends on the architecture of the target processor. Homogeneous machines need significantly less states than heterogeneous DSPs. Altogether a trellis diagram has at most

$$N_{S_{max}} = 1 + N_R 2^{N_R - 1}$$

different states [5]. In this equation, $N_R$ denotes the number of data registers. Basically the number of states increases exponentially with the number of registers. However, it can be reduced if the effects of symmetric registers are exploited. As an expample, the trellis diagrams for Motorola's DSP56000 (representing a rather homogeneous architecture) consist of 103 different states, however the trellis diagram for the ADSP2100 (which has more heterogeneous registers) contains more than half a million states, making it practically impossible to build trellis diagrams for this target architecture.

The trellis trees are built by replacing each node of the expression tree with the appropriate trellis diagram. By doing this the arithmetic operation represented by the expression tree is replaced with the corresponding machine instruction. As it cannot always be guaranteed that the destination register of one instruction is available as a source register for the following one, move diagrams have to be inserted between each two arithmetic instructions. These move diagrams also contain nop instructions (if the same register can be reused) and load/save instructions to be able to store intermediate results in the DSP memory. An example of an expression tree and the resulting trellis tree for the expression $x = c1 + c2 * c3$ is given in Figure 2. Cost values are assigned to all edges in the trellis tree. Usually the cost value is proportional to the execution time of the instruction. As most DSPs execute one instruction per clock cycle, all cost values are set to 1. In one single bottom-up traversal the path with minimum costs may be identified with the total cost value related to the run time of the generated code. The resulting sequence of instructions is the optimum straight-line code for the expression tree.

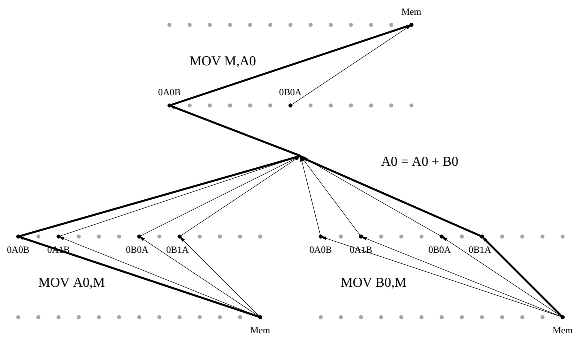

Figure 4: Dynamic trellis diagram

## 3. DYNAMIC TRELLIS DIAGRAMS

If static trellis diagrams are used to apply the algorithm described above, typically a large number of states remain unsued, especially for architectures with a heterogeneous instruction set. Typically, arithmetic or logical instructions write back their result into one or two specific registers (or registers of a specific register bank). It is therefore not necessary to create and use all possible states. The algorithm is restricted to states which are allowed by the machine description for the surrounding instructions. As most DSP architectures support more different source registers than destination registers, most of the savings will be due to the target register set of the instructions used.

The downside of this procedure is that no prebuilt diagrams may be used any more as the actual decision which and how many states have to be used depends on the current context of the instruction, i.e. the source registers of the next and the destination registers of the previous instruction. So the trellis diagrams are built dynamically whenever a node of the expression tree is encountered.

As explained before, the dynamic creation saves a large part of data memory within the compilation process, as only a small fraction of the possible states is used. Figure 4 shows an example for a trellis diagram representing an add operation surrounded by three move diagrams. By comparing this to Figure 3, the reduction of states is obvious, especially if considering that Figure 4 contains three additional move instructions which have been omitted in Figure 3.

The dynamic creation saves memory but more CPU time is used as every diagram has to be built from scratch. The consequence is a serious run-time overhead when compared to the algorithm using STDs. To reduce this overhead to a minimum, every state encountered is saved in an AVL tree. If the very same state is used again (which is likely to happen as due to the symmetric nature of most signal processing algorithms similar combinations of arithmetic instructions occur frequently), the old object will be copied instead of building it once more. This compensates some of the negative run-time aspects but not completely as there still are many different variants of trellis diagrams for each instruction. The principle of DTDs as well as measurements about their performance have been presented in more detail in [7].

## 4. EXTENDING TO COMPLEX MACHINE INSTRUCTIONS

Standard trellis diagrams do not support complex instructions as there is a one to one mapping between the arithmetic operations used in the expression tree and the diagrams used in the trellis tree. As each complex machine instruction may as well be calculated by two (or more) distinct basic machine instructions, complex diagrams do not fit into this procedure.

Dynamic trellis diagrams are created just by the time they are needed and they contain only the states which are admissible in the current context. Therefore, a strictly organized tree structure is not needed for DTDs, it is enough, if the expression tree is mapped into a directed graph (i.e. there may not be any loops).

This modification makes it possible to generate trellis diagrams not only for the atomic operations of the destination platform but also for compound commands, like e.g. MAC operations. These diagrams which may as well cover two or more levels of the expression tree. It is also possible that they have more than two source operands and they may contain edges originating from more than only one specific machine instruction. However, it is necessary that each operand corresponds to one edge of the expression tree and that each edge of the trellis diagram belongs to a certain instruction of the target hardware.

Opposed to static diagrams, where only copies of a template are placed into the new tree, with DTDs it is not possible to use a simple top-down method to create the trellis tree. However, it is necessary to keep track of the operands whenever a diagram with a height of two or more levels is processed as the same operands will be used later on when the appropriate basic diagrams are inserted. The best place to store such information is the expression tree, as in its structure it corresponds to the new trellis tree. So it is necessary to add new attributes to the nodes of the expression tree pointing to dangling operands of the trellis tree which is just being built. It is important that all dangling operands are resolved during the building process. If any end remains unconnected, the resulting tree is not valid.

## 5. EXPERIMENTAL RESULTS

The part of an existing code generator containing the DTD creation and manipulation has been extended to support complex machine instructions. The trellis tree has been modified to contain temporary information during the time the trellis tree is built. The existing target description language has been extended to support complex machine instructions by concatenating two or more simple arithmetic operations to one command. At present, descriptions for Motorolas DSP56000 and Analog Devices ADSP21000 contain the description of MAC instructions. An extension to support rounding and saturation operations is not complicate and planned for the future. The program has been tested with several different examples and the results have been examined. It is not possible to give generic results, as the savings and the quality of the resulting code heavily depends on the actual graph. Applications which offer more opportunities for the application of complex machine instructions will benefit more than small programs which don't. Basically, every potential MAC operation will be exploited. An example, where the savings are big is a second order state space filter, which needs 10 instruction cycles with MAC instructions, but 16 without.

As the same data structures are used and the time-critical algorithms are not changed the extension presented in this paper does not change the memory or CPU requirements of the algorithm. The only difference is a broader selection of machine instructions to choose from, but as there are only a few complex instructions opposed to a variety of different add instructions (to support different target register set, memory banks or addressing modes), the impact on the run-time behaviour is not noticeable.

Opposed to STDs, a moderate run-time increase can been observed, resulting from the creation of every single trellis diagram opposed to the usage of precalculated diagrams for the STD algorithm. This increase is partly but not completely compensated by a speed gain during the evaluation of the trellis tree, resulting from the use of a much smaller number of nodes and links. The total run-time strongly depends on the degree of orthogonality of the architecture, ranging from a few seconds for Motorola's DSP65000 series to several minutes for Analog Devices ADSP2100. At this point it has to be noted, that due to the huge number of states for the ADSP processors, it was not possible at all to compile code for these architectures using STDs as well as to use arbitrary complex machine instructions for any architecture. Therefore the increased run-time is not avoidable if code for heterogeneouse architectures has to be generated.

## 6. CONCLUSIONS

The proposed method of using DTDs to generate highly optimized DSP code makes DFG based code generation applicable to most available general purpose DSP architectures. By omitting unnecessary nodes and edges in the trellis tree, the amount of memory needed can be significantly reduced. This can be done by creating new trellis diagrams for each and every node in the tree instead of using precalculated information. When creating the diagrams, only nodes which are actually being used are added to the trellis diagram. The savings are especially high for non-orthogonal DSP architectures.

Further savings can be achieved by supporting the usage of complex machine instructions. These instructions are difficult and awkward to support with static trellis diagrams. Programs which contain multiplications followed by additions would require two different trellis diagrams, one containing the individual atomic operations and another one for the complex MAC instruction. This is not possible within the scope of STDs, so workarounds have to be used which cannot be generalized. The proposed algorithm allows to exploit MAC- and rounding operations as well as other, machine specific complex arithmetic operations in a very general way. It is able to find the optimum solution for each graph which may be represented within one expression tree. The complex machine instructions are defined in the target architecture description file and applied to the input DFG. Complex instructions are used whenever they lead to shorter (and therefore faster) machine code. It is also possible, although not yet implemented, to map machine instructions of a different arithmetic or logical type into a single trellis diagram, whenever this is permitted. As an example it may be shorter to perform a multiplication by 2 by using an add instruction and adding the operand to itself. As a third possibility, a shift operation by one bit is also possible. All these instructions can be combined into one dynamic trellis diagram, the standard algorithm will select the best alternative which is used in the final assembly code.

The code generator is architecture independent, the assembly language may be specified using a special hardware description language. This specification is used for the dynamic creation of the trellis diagrams. Complex instructions have to be specified here to be recognized while the trellis diagrams are generated. The usage of instructions of a different arithmetic or logical type requires no specification in the architecture description but has to be integrated into the trellis diagram generation process.

## 7. REFERENCES

[1] M. Gotschlich and B. Wess, “Automatic generation of constrained expression trees for global optimized DSP assembly code”, *in Proc. 7th Int. Conf. on Signal Processing Applications & Technology*, vol. 1, pp. 732–736, Boston, October 1996.

[2] B. Wess and W. Kreuzer, “Optimized DSP assembly code generation starting from homogeneous atomic data flow graphs”, *in Proc. 38th Midwest Symp. on Circuits and Systems*, vol. 2, pp. 1268–1271, Rio de Janeiro, August 1995.

[3] M. R. Garey and D. S. Johnson, *Computers and Intractability*, W. H. Freeman and Company, 1979.

[4] B. Wess, “Automatic instruction code generation based on trellis diagrams”, *in Proc. IEEE Int. Symp. on Circuits and Systems*, vol. 2, pp. 645–648, San Diego, May 1992.

[5] B. Wess, “Code generation based on trellis diagrams”, in P. Marwedel and G. Goossens, editors, *Code Generation for Embedded Processors*, chapter 11, pp. 188–202. Kluwer Academic Publishers, 1995.

[6] U. Krebelder, C. Brem, S. Fröhlich, and M. Gotschlich, “Target description language TDL - Ein Dateiformat zur Beschreibung von Signalprozessoren”, Technical Report VCGRG-97-1, Institut für Nachrichtentechnik und Hochfrequenztechnik, University of Technology, Vienna, 1997.

[7] S. Fröhlich, M. Gotschlich, U. Krebelder, and B. Wess, “Dynamic trellis diagrams for optimized DSP code generation”, *in Proc. IEEE Int. Symp. on Circuits and Systems*, Orlando, June 1999.