DYNAMIC TRELLIS DIAGRAMS FOR OPTIMIZED DSP CODE GENERATION

Stefan Fröhlich, Martin Gotschlich, Udo Krebelder 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). In contrast to static trellis diagrams (STDs), which may be precalculated, DTDs are built at runtime and adapted exactly to the local requirements. Therefore, DTDs are more flexible and need less program memory. Due to the significant reduction in memory size, the 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

The programming of general purpose DSPs is either very inefficient or very time-consuming. The reason for this behaviour is the lack of well performing compilers in an environment where real-time algorithms have to be implemented. Typically, hand-coded DSP programs are about a magnitude faster than code which is generated from a standard DSP C-Compiler.

To produce better results, a DFG compiler has been developed which is able to produce highly optimized assembly code from a DFG specification. An overview of the compiler is given in Figure 1. It consists of a DFG decomposition unit which cuts the DFG into one or more expression trees (ETRs) [1, 2].

This step is necessary because the generation of optimal code directly from DFGs is known to be NP-hard [3]. An example for a simple expression tree and a trellis tree is given in Figure 2. The next step is to generate locally optimal code for each of these trees using the proposed algo-

This work was supported by the Fonds zur Förderung der wissenschaftlichen Forschung (FWF) under research grant P10701-ÖTE

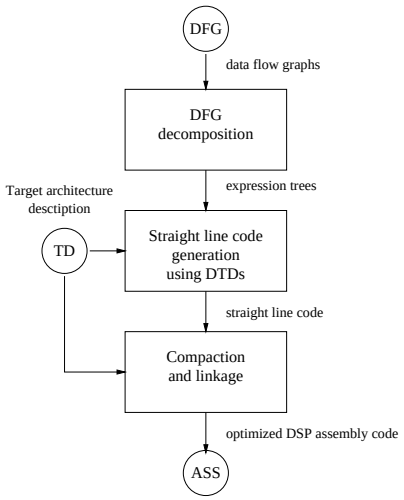

Figure 1: Basic blocks of the DFG compiler

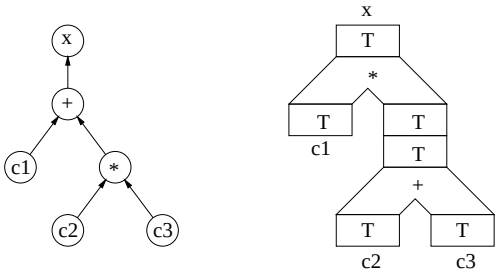

Figure 2: Exptression tree and trellis tree for $x = c1 + c2 * c3$

rithm, which has an $O(n)$ run-time behaviour [4, 5]. During this step, a trellis tree is generated by concatenating the appropriate trellis diagrams and inserting transfer diagrams in between. As the generated code is optimal within each expression tree, the trees should have the maximum size possible. Afterwards, the code segments are concatenated and compacted by combining two or more instructions into one whenever this is possible, yielding highly optimized code for the given 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

Previous versions of the DFG compiler have been using static trellis diagrams. Trellis diagrams are the building blocks for trellis trees which serve as an underlying data structure in the code generation process. As shown in Figure 3, they consist of nodes and edges. The nodes, which are also called states, correspond to a particular storage resource type which can either be a specific register or an arbitrary memory location. Each state specifies not only the operand location l_k but also a set of data registers R_k which is assumed to be available for the execution of the instructions. Edges correspond to instructions of a specified arithmetic or logic operation.

The trellis diagram presented in Figure 3 shows the relevant parts of the add instruction $A = A + B$ of a hypothetical four-register machine with the registers A_0, A_1, B_0 and B_1. At the top of the figure, the destination states are listed. In this diagram we only consider state *0A0B* which indicates that none of the registers is locked and the result is stored in register *A*. In our notation, writing a register first means that it contains the current result. Registers A_0 and A_1 are symmetric, so the algorithm only has to decide in which bank the result is stored. The exact register may be determined later on. The two lines at the bottom represent the register states for the operands. If the left operand is evaluated first, again all registers may be used and the operand is stored in register *A*. The right operand, which is evaluated afterwards, may not modify the register with the intermediate result evaluated before. Thus state *0B1A* is used, indicating that 1 register of set *A* is locked and the result is stored into register *B*.

The dotted pair of lines is used if the right operand is evaluated before the left one. Additionally, the statement could be rewritten $A = B + A$, thus swapping the two trees,

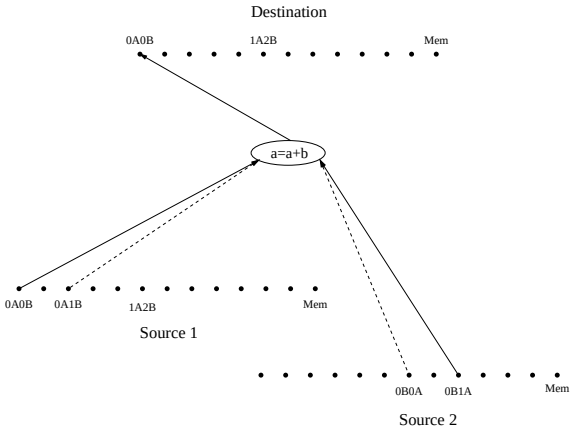

Figure 3: STD of a simple add instruction

which would lead to two more sets of states. However, the principle remains the same, so these lines are skipped in the diagram. If one or more registers are locked because of the instructions above the current one, the appropriate state has to be used for the destination register. The procedure to derive the source states remains the same.

A trellis diagram has altogether at most

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

different states [7]. In this equation, N_R denotes the number of data registers. Essentially 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 (having more symmetric registers) 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.

Trellis trees are built from exptression trees by replacing each ETR node by a trellis diagram. The arithmetic or logic operation of the ETR node corresponds to the instructions of the trellis diagram. The trellis diagrams are augmented with move diagrams which ensure that the instruction operands of adjacent arithmetic or logic operations may be located in different registers, or that intermediate results may be stored in memory. An example of an expression tree and the resulting trellis tree is given in Figure 2 for the expression $x = c1 + c2 * c3$. Cost values are assigned to all edges in the trellis tree. The cost of an edge is typically the number of processor cycles necessary to execute the instruction. 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

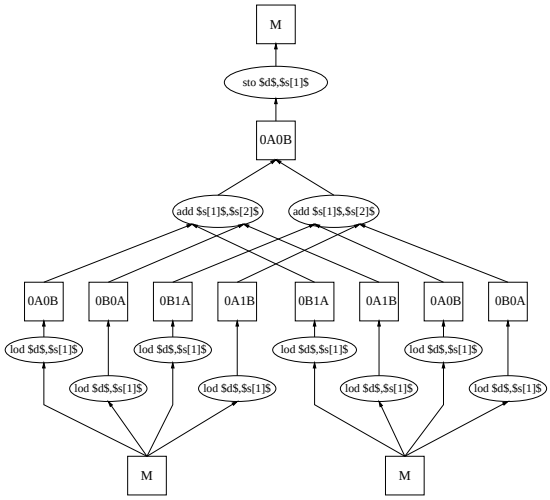

Figure 4: Dynamic trellis diagram

of instructions is the optimum straight-line code for the expression tree.

3. DYNAMIC TRELLIS DIAGRAMS

When using the algorithm described above, it can be noticed that often a large number of the states remain unused. Typically, an arithmetic or logical instruction writes back the result into a certain register or register bank. In these cases it is unneccessary to create all possible states (which are permutations of all registers currently available). We restrict the algorithm to states which are candidates for the optimum path. The same is true for the source operands. However, in typical DSP architectures, there are more different source registers available than destination registers, so the savings will be smaller. To use these potential savings, it is no longer possible to create prebuilt trellis diagrams, as the actual number of states used depends on the current context, i.e. the set of possible destination registers of the instruction right before and the possible source registers of the next instruction. Instead, the trellis diagrams are built dynamically whenever a new node of the expression tree is encountered.

The dynamic construction saves program memory, as only a small fraction of all possible states will be used. For example in Figure 3 all the nodes without an edge leading to them would be skipped. An example for a DTD is given in Figure 4, which shows the dynamic trellis diagram for an addition augmented by move diagrams. On the other hand, every trellis diagram has to be created from scratch, resulting in a run-time overhead when compared to the prebuilt diagrams of the STD algorithm. To reduce this overhead, every state encountered is saved into an AVL tree. If

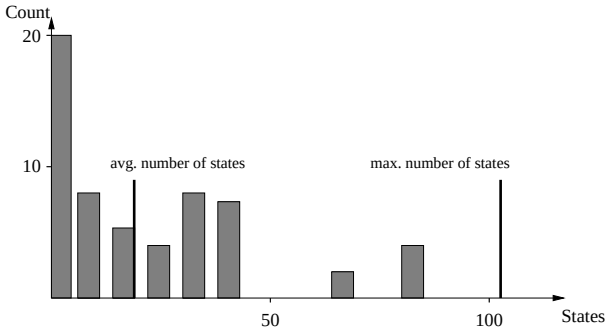

Figure 5: Number of states used to generate DSP56k assembly code

a specific state is used another time (which is likely to happen due to the symmetric nature of most signal processing algorithms) the old state objects can be copied instead of building them another time which compensates some of the negativ run-time aspects.

4. EXPERIMENTAL RESULTS

DTDs have been integrated into a code generator transforming DFGs into generic DSP assembly code automatically. The program has been tested with several different graphs and the results have been examined. It is not possible to give an exact calculation concerning the savings due to the use of DTDs because the actual number of links used heavily depends on the graph which is parsed. However, note that the reduction of memory usage is significant as can be seen in Figures 5 and 6 which have been generated from a DFG describing a second order lattice filter. The DS56000 diagrams are significantly smaller due to the symmetric nature of the processor. The savings compared to a STD are about 75% (the average number of states is approximately 21 opposed to a maximum number of 103). The ADSP2100 diagrams are one magnitude larger, however they are more than 99% smaller than the corresponding STDs. This is due to the highly asymmetrical architecture of the ADSP2100 which makes the use of STDs practically impossible. The average number of states with DTDs is 275.

A moderate run-time increase has 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 compensated by a speed gain during the evaluation of the trellis tree, resulting from the use of much lesser nodes and links. The total run-time strongly depends on the orthogonality of the architecture, ranging from a few seconds for Motorolas DSP65000 series to several minutes for Analog Devices ADSP2100. It has to be noted, that due to the huge number of states for the ADSP processors it was impossible to compile code for these

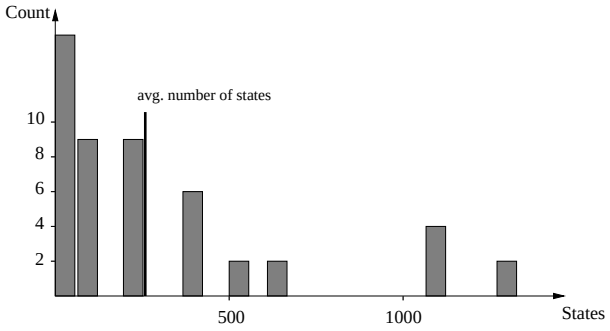

Figure 6: Number of states used to generate ADSP2100 assember code

architectures using STDs.

5. CONCLUSIONS

The proposed method of using dynamic trellis diagrams 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. The code generator is architecture independent, the target language may be specified using a special hardware description language. This information is used for the dynamic creation of the trellis diagrams. Besides, the much more efficient memory usage, DTDs allow other improvements of the code generation process, which have yet to be exploited.

For the scope of this paper, DTDs have been used to reduce storage when building and evaluating the trellis tree. However, it seems to be possible to gain additional benefits. With STDs, the implemenation of multiple instructions like the MAC command, which is typical on standard DSP architecures, has shown to be problematic. As a program may either contain a multiplictation command followed by an addition, or on the other hand may use a single MAC instruction, one would need two different kinds of STDs to cover the trellis tree. As this is not possible, workarounds like virtual registers have been introduced. These workarounds are awkward and cannot be generalized. With DTDs however, there is the possibility to make a dynamic connection directly from the leaves of the MAC instruction to the register containing the result. This may be used in combination with the standard connections of the separate multiplication and addition. Effectively the two alternatives are both present in the trellis tree (without the necessity of any virtual registers), allowing the compiler to choose the more efficient one. Another improvement is the possibility to map different types of instructions to one single node of the trellis diagram. For example, a multiplication by two may be expressed as a multiplication (with a factor two), an addition (with equal operands), or as a shift command (by one to the left). With STDs, these additional opportunities could not be exploited as the trellis diagrams for the individual commands have been prebuilt and could not be adapted to the local requirements. DTDs have this capability, as they are created at a point where more detailed information about the input operands is available.

6. 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*, volume 1, pages 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*, volume 2, pages 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*, volume 2, pages 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, pages 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] M. Gotschlich. Automatischer Codegenerator für die Signalprozessorfamilie ADSP-2100. Diplomarbeit, October 1995.