Generating a CDFG from C/C++ code

Michèl A.J. Rosien, Gerard J.M. Smit, Thijs Krol
Department of Computer Science,
University of Twente, Enschede, the Netherlands
email:rosien@cs.utwente.nl

***Abstract*—This paper presents a method to automatically generate a Control Data Flow Graph (CDFG) from C/C++ source code. This CDFG is used to automate the programming of an FPFA, a flexible, energy efficient reconfigurable device, introduced in the Chameleon project.**

I. Introduction

The Chameleon project introduced the Field Programmable Function Array (FPFA), a flexible, energy efficient reconfigurable device with a datapad that can be reconfigured to implement a number of Digital Signal Processing (DSP) algorithms. Energy efficiency is achieved by locality of reference, while performance is achieved by parallelism. Due to this parallism, it is difficult, time consuming and error prone to manually program the FPFA. Our goal it to automate the programming of the FPFA, and to be able to program the FPFA using C/C++. This is achieved by first transforming the source code into a Control Data Flow Graph (CDFG). This CDFG can then be scheduled onto the FPFA more easily. For more information about the FPFA see [2].

This paper presents a method to automatically generate a CDFG from C/C++ source code.

II. Definition of a CDFG

A Control Data Flow Graph (CDFG), is a graph that represents the operations (for example the C/C++ operators and function calls) and the dataflow between those operations. The dataflow can be normal 'data' such as operands of mathematical operations, the statespace (see sectrion V) or control information which is used to translate the selection and iteration statements of C/C++. Examples of such statements are: the **if** statement and the **while** statement.

A CDFG is modelled using the *hyper*graph model. In conventional graphs, operations are represented by nodes, and the inputs and outputs are represented by the edges between those nodes. In the hypergraph model, the operations are represented by the edges (called *hyper*edges) and the inputs and outputs are represented by the nodes which connect the edges. This way, an operation can have any number of distinguishable inputs/outputs. Figure 1 shows an example of a hypergraph consisting of two hyperedges and a number of nodes, representing two add operations.

An advantage of this method is that a hypergraph itself can be used as a definition of a new hyper*edge*. This way, a hierarchical graph can be created. The translation process uses this hierarchy to put certain C/C++ constructs in different hierarchies. For example, a compound statement is put in a different hierarchical level than the surrounding statements. This hierarchical approach simplifies certain transformations such as loop unrolling. For more information about the hypergraph model see [1].

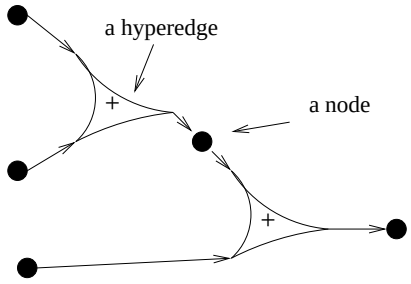

Fig. 1. Example of a small hypergraph.

III. Approach

The process of generating a CDFG from C/C++ code is split up into several steps. Figure 2 shows these steps. The first step is to generate a parse tree of the code using a C++ parser frontend. Once the parse tree is generated, the language constructs, like expressions and *for* statements, become clearly visible. The language constructs found in the parse tree are then converted into a list of hypergraph *templates*, which are explained in section IV. Using the list of templates, the complete CDFG is built. The resulting CDFG is a completely hierarchical hypergraph representation of the C/C++ code. A predefined set of behaviour preserving transformations can be applied to the graph to try to obtain a "clean" graph. A "clean" graph is a graph

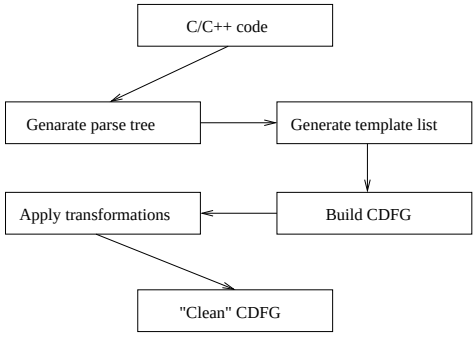

Fig. 2. Generating a CDFG from C/C++ code.

where the statespace (see section V) and control lines (see section VI) are removed as much as possible. After obtaining a "clean" graph, additional transformations can be applied to try to create a graph which "fits" better onto the target architecture (FPFA).

IV. Hypergraph templates

A hypergraph template, is a parameterizable hypegraph. In other words, a hypergraph that needs other hypergraphs as parameters before it is fully defined. An example of a hypergraph template, hereafter called 'template', is the *BinaryExpression* template, which is used to build graphs that correspond to expressions of the form: <expression> <binaryoperator> <expression>. For example: "a - 6". This template needs three parameters, two expression graphs (which can be templates themselves) and a binary operator graph.

To make sure the parameter graphs "fit" into the template, the template only accepts graphs of a certain class. The *BinaryExpression* template, for example, only accepts parameters of the *expression* class. Graphs that belong to the same class have the same number of inputs and outputs. This approach ensures that the number of inputs and outputs of the parameter graph matches the number of input and outputs of the parameters needed by the template.

Currently, templates have been designed for almost every C/C++ construct except classes and exceptions. Approximately 25 - 30 templates have been designed. Exceptions will be included in the future. It is not yet clear how to map classes on a hardware architecture.

V. Representation of the C/C++ memory model

To be able to map C/C++ to a CDFG it is necessary to have a way to represent the C/C++ memory model in the hypergraph model. For this purpose the *statespace* is introduced. The statespace is a mathematical abstraction of the memory of a C/C++ program. The memory is modeled as a set of tuples, which correspond to the variables and their values. A statespace with two variables a and b, with the values 3 and 5 respectively, is modeled as: $\{(a, 3), (b, 5)\}$.

Three primitive hypergraphs that are able to interact with the statespace are introduced. These graphs are:

1) *store* : Writes a tuple to the statespace.
2) *fetch* : Reads a tuple from the statespace.
3) *delete*: Deletes a tuple from the statespace.

Since the complete memory is modeled by one global statespace, there is no distinction between different scope levels. To still be able to support scopes, i.e. variables with the same name in different scopes, variables have to be renamed so that they are unique in the global memory. With this simple memory model also pointers, arrays and structs can be described.

VI. Representation of control flow

One important issue with the CDFG approach is how control flow has to be implemented, since it is not possible to "jump" from one point to another point in a CDFG. Our solution is to implement iteration and jump statements using recursion (selection statements like the **if** statement can be modeled by splitting the statespace using a mux).

Implementing jump statements (goto, return, break and continue) using recursion deserves some more explanation. Since jumps are not directly supported in a CDFG, jumps are modeled by *repetition* combined with control lines which determine whether certain statements are executed or not. When a jump is encountered, all statements are skipped until a matching destination label is found. A jump backwards is handled by the fact that the complete block in which the jump statement resides is repeated using recursion.

VII. Example of a generated CDFG

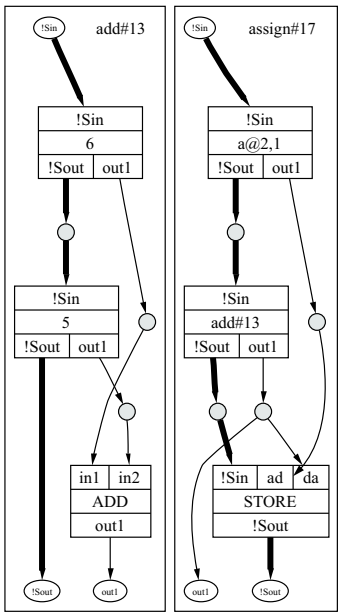

Fig. 3. A CDFG of the expression a = 6 + 5.

Figure 3 shows an automatically generated CDFG of the expression $a = 6 + 5$. The left graph is the binary sub-expression $5 + 6$ (called *add#13* in this figure). This graph has two sub expression graphs called *6* and *5* and a binary operator graph called *add*. The bold line is the statespace which goes through both expressions since expressions in C/C++ might have side effects. In this case they don't have side effects since the expressions are constants. But the binary expression template graph requires that the statespace always goes through its subexpressions.

The right graph is the assignment expression $a =$<*add#13*>, where <*add#13*> is defined by the left graph. Since the left hand side of a C/C++ assignment can also be an expression, the statespace has to go through the id expression ($a@2, 1$) as well. The numbers 2 and 1 in the id expression $a@2, 1$ are numbers added to the variable a to make it unique on a global scope level. These numbers are a representation of the scope in which this variable is found.

Figure 4 shows the same expression as was shown in figure 3 but with the hierarchy removed. The statespace now only passes through the *store* block.

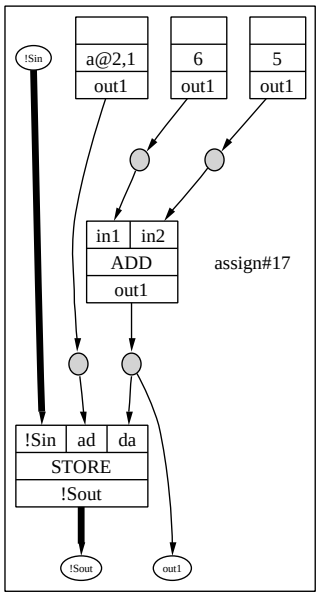

Fig. 4. The graph of figure 3, with hierarchy removed.

VIII. Conclusion

The presented method is able to generate a CDFG from a subset of the C/C++ language. This CDFG can then be mapped onto the target architecture(FPFA). Behaviour preserving transformations, such as loop unrolling, constant propagation or dead code elimination, can simplify this mapping. The target architecture can be extended to a Field Programmable Gate Array (FPGA), a heterogenous reconfigurable architecture. Our model supports a large number of C/C++ constructs including pointers, which similar tools (such as Art Designer) do not support. Classes, however, are not supported yet. It is unclear how to map classes on a hardware architecture.

Acknowledgements

This research is conducted within the Chameleon project (TES.5004) supported by the PROGram for Research on Embedded Systems & Software (PROGRESS) of the Dutch organization for Scientific Research NWO, the Dutchs Ministry of Economic Affairs and the technology founddation STW.

References

[1] Thijs Krol and Bert-Steffen Visser: “High-level Synthesis based on Transformational Design”, *Internal report, University of Twente*, Enschede, The Netherlands.

[2] Paul M. Heysters, Henri Bouma, Jaap Smit, Gerard J.M. Smit, Paul J.M. Havinga: “A Reconfigurable function array architecture for 3G and 4G wireless terminals” In press: *2002 International Conference On Third Generation Wireless and Beyond*, May 2002.