A Parallelizing Compiler for the Functional Programming Language EVE *

Hans Wolfgang Loidl
Research Institute for Symbolic Computation (RISC-Linz)
Johannes Kepler University, Linz, Austria
E-mail: hwloidl@risc.uni-linz.ac.at

October, 1992

Presented at: First Austrian-Hungarian Workshop on Transputer Applications, Sopron, Hungary, October 8–10, 1992

Abstract

This paper describes the compilation process from the non-strict functional programming language EVE to the assembler code of the abstract hybrid dataflow/controlflow machine ADAM. In this compilation process dataflow graphs are used as an intermediate language. However, the resulting assembler code is a controlflow program that simulates the behaviour of the dataflow graph. The non-strict semantics of EVE ensures that it contains maximal inherent parallelism. The target language of the compilation process is ADAM assembler code, which is essentially a controlflow code that is augmented with instructions for explicit creation and synchronization of parallel tasks. The compiler is written in Modula-2 and is running on a PC.

1 Introduction

Among all programming languages, functional programming languages have the advantage of containing most inherent parallelism. The main problem with these languages is to exploit this parallelism by creating efficient code for an imperative (control-driven) machine. Our approach for solving this problem is to use dataflow graphs as an intermediate language and then to create code that simulates the dataflow behaviour on an abstract hybrid dataflow/controlflow machine.

The work presented here is part of the ADAM & EVE project. The main goal of this project is to study the behaviour of programs in the functional programming language EVE that have been compiled to the assembler code of the hybrid dataflow/controlflow machine ADAM. Another goal of the project is to test the applicability of the dataflow

* *Sponsored by: Austrian Forschungsförderungsfonds, project no. P-06931-PH*

model for the automatic parallelization of functional languages in general. For executing these programs a simulator on a network of 16 transputers has been implemented. The architecture of the ADAM machine will be described in Section 2. For a detailed description of the ADAM machine, see [Schreiner, 1990].

The main part of this paper describes the results of the second part of the project, which consisted of the implementation of a compiler generating ADAM assembler code from the functional programming language EVE. The basic ideas lying behind this translation process will be discussed here.

This project has been influenced by the work at MIT on dataflow architectures and compilers for these architectures. For example, the handling of data structures is done with I-structures that are described in [Arvind *et al.*, 1987]. Similar ideas for the compilation of a functional language to the assembler code of a hybrid dataflow/controlflow machine can be found in [Iannucci, 1988] and also in [Traub, 1986].

2 The ADAM Abstract Dataflow Machine

In the frame of the ADAM & EVE project an abstract machine for the parallel execution of functional programs on a distributed memory machine has been designed. The *design principles* for this ADAM machine have been the following:

- The *dataflow model* is used for the automatic parallelization of programs in the high-level programming language EVE. The reason for choosing the dataflow model is the fact that this model shows the maximal parallelism that lies in a program as the parallelism is only restricted by the data-dependencies between the operations. However, nowadays much effort goes into using the graph reduction model for the automatic parallelization of functional languages (see [Loogen, 1990], [Peyton Jones, 1989]). Therefore, this project should also test the applicability of the dataflow model for the automatic parallelization of functional languages.
- The *controlflow model* is used for the execution of the programs. This means that no special hardware is designed in this project. Instead, stock hardware is taken, thereby profiting from the increasing speed of conventional hardware.

The conclusion of the above two design principles is to **simulate dataflow behaviour on the controlflow model**.

The global structure of the ADAM machine is formed by several independent *worker modules*, which can be connected in a any topology. Hence, the machine is based on the distributed memory principle.

The structure of one of these worker modules can be seen in Figure 1. There one can distinguish the three main components of a worker module:

- The **arbiter**,

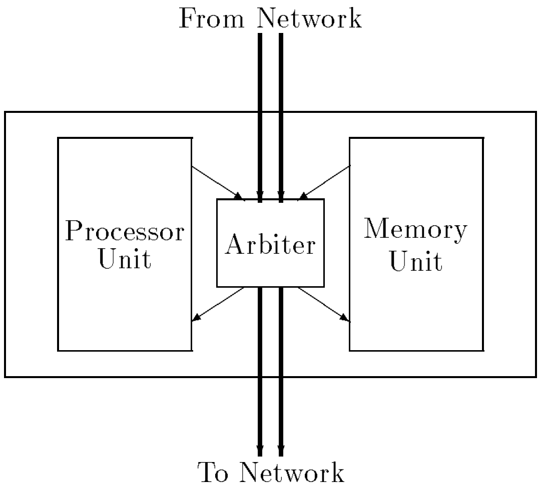

Figure 1: Structure of an ADAM Worker Module

- the **memory unit** and
- the **processor unit**.

These three components will be described in more details in the following sections.

2.1 The Arbiter

This component is responsible for handling the communication with other modules in the network. Therefore, this unit is the interface of one worker module to the others in the network.

2.2 The Memory Unit

This component is responsible for the handling of non-strict data structures. The difference between a strict and non-strict data structure is that the latter can be used even before all of its components have been computed. From the parallelism point of view this is a very important property as this allows a straightforward usage of the so called producer-consumer parallelism. In this kind of parallelism a producer process creates a usually quite big data structure and a consumer process performs operations on this data structure. Figure 2 shows an example where the producer creates a list that is the input for a consumer process. Especially when the consumer process also creates a list it does not need the values of all components in the data structure to compute one component of the resulting list. In a language with non-strict data structures both processes can work in parallel as the consumer does not have to wait for the production of the whole data structure as it would be the case in a strict language. Therefore, one

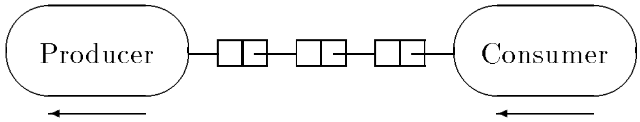

Figure 2: Producer-Consumer Parallelism

gets the picture in Figure 2 describing a race of two parallel processes from the right to the left where the consumer process may never overtake the producer process.

For the implementation of non-strict data structures so called I-structures are used, which have been introduced in the frame of a project realizing the tagged token dataflow architecture at MIT (see [Arvind *et al.*, 1987]). In these I-structures every component of a data structure has a full/empty bit associated with it. Due to the single assignment rule in functional languages a process can only write into a component whose bit is set to empty. If a process tries to read an empty component this read request is deferred and stored in a queue that is associated with this component. The process that has initiated the read request will be automatically blocked until another process writes a value into this component. As soon as this writing is performed, the deferred read requests are fulfilled and the blocked processes are reawoken. By this principle, non-strict data structures are the main medium of communication between parallel processes.

2.3 The Processor Unit

The processor unit of an ADAM module is responsible for performing the actual computation in an ADAM worker module. To explain the assembler code of the ADAM machine take the following mathematical expression:

$$z = -x + x * y$$

This expression can be straightforwardly translated into the dataflow graph in Figure 3. In this dataflow graph the nodes are labelled for better reference. From this graph the implicit parallelism in the program becomes obvious. For example the operations labelled with neg and mul can be performed in parallel as there is no data dependency between them. However, the addition operation needs the result of the multiplication and therefore these two operations can not be performed in parallel.

The dataflow graph of Figure 3 can be translated into pure dataflow code by taking each node in the graph and translating it into one line of code consisting of the label, the operation and the successor instruction (in the case of the copy node two successor instructions are necessary). This translation yields the dataflow program in Figure 4.

The problem with the pure dataflow code is that in general many copy instructions are needed to copy an input value as many times as it is needed in the dataflow program. In bigger programs this yields rather inefficient code. Therefore, in the ADAM machine

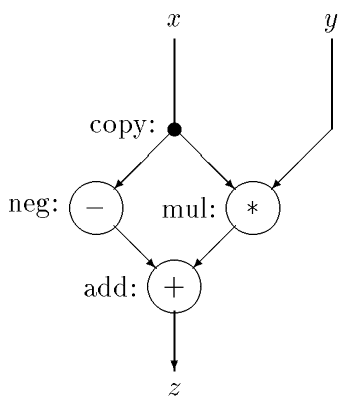

Figure 3: Dataflow Graph of $z = -x + x * y$

copy:	COPY	neg, mul.l
mul:	MUL	add.r
neg:	NEG	add.l
add:	ADD	result

Figure 4: Pure Dataflow Program of $z = -x + x * y$

a different approach has been taken. First of all, the well known notion of a register has been introduced. The arithmetic operations work on these registers. By this, one gets a conventional controlflow code with two address instructions and explicit jumps. However, to handle parallelism two additional constructs have to be introduced:

- The SPLIT instruction, that splits the controlflow into two parallel threads. From the dataflow point of view, this instruction represents a node that takes one input token and splits it into two independent output tokens like the copy node in Figure 3. This construct is used to *create parallelism.*

- The WAIT instruction, that joins two threads in the controlflow into one thread. From the dataflow point of view, this instruction represents a node that takes two input tokens and merges them into one output token. This construct is used to *synchronize parallel parts* of the program.

Figure 5 shows the ADAM assembler program that corresponds to the dataflow graph in Figure 3. In the first two instructions of the program the input values are put into the registers x and y. The second argument of the RECEIVE instruction denotes the successor instruction. In the case of the first input value a splitting into two parallel threads of control has to be performed. One thread negates the values in x and stores

```
        RECEIVE   x, split
        RECEIVE   y, mul
split:  SPLIT     neg, mul
neg:    NEG       z, x, add
mul:    WAIT      e1, +1
        MUL       x, y, add
add:    WAIT      e2, +1
        ADD       z, x, result
```

Figure 5: ADAM Assembler Code of $z = -x + x * y$

the result into the register z, the other thread jumps to the multiplication operation. However, at both binary operations first of all an explicit synchronization of the input values has to be performed. This is done in the WAIT instruction that takes an event as its first argument. This event is initially set to "not occurred". As soon as the first thread arrives at the WAIT instruction the event is set to "occurred" and this thread is blocked. When the next thread arrives the event is set to "not occurred" and exactly one thread continues the computation with the instruction denoted by the second argument of the WAIT instruction. Note that this synchronization scheme was implicitly present in the evaluation semantics of the dataflow graph and has become explicit in the ADAM assembler program.

The *advantages of* ADAM *assembler code* compared to pure dataflow code can be summarized as follows:

- The data is shared in registers and therefore fewer copy instructions are necessary.
- The creation of parallelism and the synchronization are explicit and can be changed by the compiler, thereby choosing another granularity of the parallel processes than it is suggested by the dataflow graph itself.

3 The EVE Compiler

After having described the details of the ADAM machine and its assembler code, in this section the compilation process from a functional program to the assembler code of the ADAM machine will be discussed with respect to the parallelism that is extracted out of the program. This process is split into two parts with dataflow graphs as the intermediate language. In this paper we will concentrate on only one construct namely the function call. However, the most important aspects of the translation of a functional language construct to a dataflow graph and then to ADAM assembler code can already be shown with this construct.

Before we go into the details of the compilation itself we have to take a closer look at the intermediate language of the translation process. As it has been mentioned above

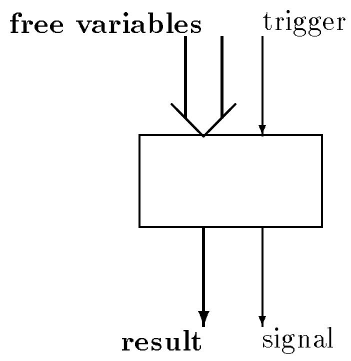

Figure 6: Interface of a Dataflow Graph

this intermediate language is formed by dataflow graphs. However, for translating all functional language constructs into such dataflow graphs two special arcs are needed in the graph:

- The *trigger* arc is an additional input arc that is needed for starting a non-strict function even before the first of its arguments has arrived.
- The *signal* arc is an additional output arc that is needed to guarantee that no token is left in the dataflow graph when tokens have appeared on every output arc (*self-cleaning property*). Without this arc, the self-cleaning property would be lost for dataflow graphs that contain side-effect nodes.

In Figure 6, which shows the interface of a dataflow graph, the trigger and signal arcs are drawn thinner than other arcs as the tokens on these arcs do not carry meaningful values. Just the arrival of a token on the trigger or on the signal arc delivers some important information.

3.1 From a Functional Program to a Dataflow Graph

One of the most interesting constructs of a non-strict functional language with respect to its implicit parallelism is the function call. The dataflow graph for this construct is shown in Figure 7. Obviously, all the arguments of the function call can be evaluated in parallel. However, due to the non-strict semantics of the language, the function that is called can be evaluated in parallel with its arguments. When in the evaluation of the function the value of an argument is needed that has not been computed yet, the evaluation of the function will be suspended.

Another consequence of the non-strict semantics is the necessity of splitting the invocation of a function and the sending of the arguments to this function into two parts.

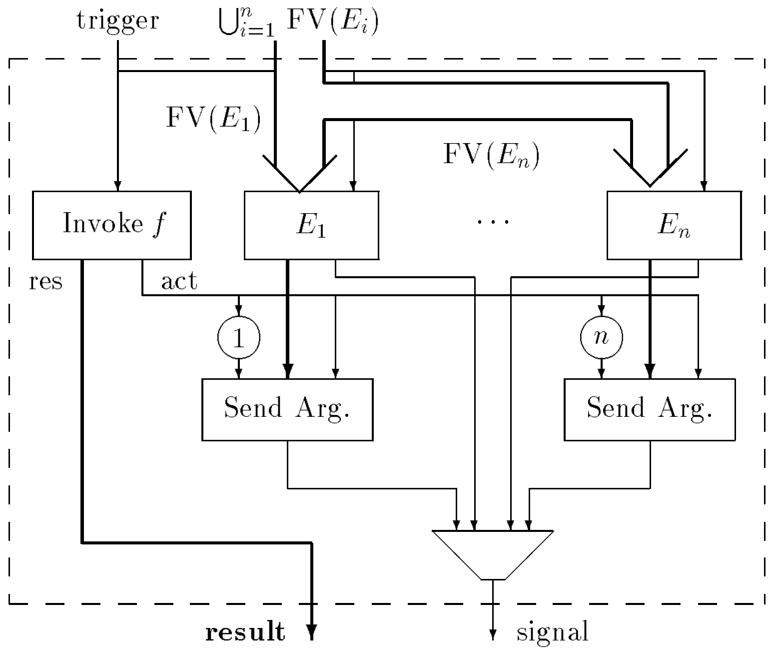

Figure 7: Dataflow Graph for the Function Call: $f[E_1, \ldots, E_n]$

As an argument can only be sent to a function after having invoked this function, a synchronization between function invocation and the sending of arguments has to take place.

The nodes for the invocation of a function and for the sending of an argument are special in the sense that they perform some side-effects. For a further description of these side-effects see Section 3.2.

3.2 From a Dataflow Graph to an Assembler Program

As it has been described in Section 2.3, the ADAM assembler code is essentially a conventional two address code in which almost all instructions contain a reference to their successor instructions at which the computation will be continued. Usually the successor is addressed via a relative distance (written as a signed number). However, one can also use symbolic labels instead of relative distances. The instructions SPLIT and WAIT are used for the creation of parallelism and for synchronization, respectively.

	$\mathbf{C}_{split,trigger}$		; Feed trigger to fct and
			; to each argument
$invoke$:	FASTCALL	$f, +1$	; Invoke function
act :	RECEIVE	$r_{act}, send$	; Fct activation arrives
res :	RECEIVE	$r, result$	; Result of fct arrives
	$\mathbf{C}_{split,send}$		; One send for
			; each argument
	$\mathbf{C}_{E_i}$		; $i = 1, \ldots, n$
$result_{E_i}$:	WAIT	$e_i, +2$	; Synchronize result of
$send_{E_i}$:	WAIT	$e_i, +1$	; arg. with fct. invocation
	SEND	r_{E_i}, r_{act}, i	; send ith arg. to fct.
	JUMP	arg_{E_i}	
	$\mathbf{C}_{SignalTree}$		

Figure 8: ADAM Assembler Code for the Function Call: $f[E_1, \ldots, E_n]$

Figure 8 shows the assembler code that is produced from a function call whose dataflow graph has been shown in Figure 7. For clarification of the main aspects of the code, the distribution of the free variables among the expressions E_i $(i = 1, \ldots, n)$ is not shown here. The input and the output of the whole graph are the following labels, which correspond to arcs in the dataflow graph:

- **Input:** *trigger*.

- **Output:** *result*, *signal*.

The code sequence $\mathbf{C}_{split,trigger}$ splits the controlflow into n+1 parallel threads. One thread invokes the function f and the other threads start the computation of the argument expressions E_i. This corresponds to the splitting of the trigger token into n+1 successor tokens in the dataflow graph. The interface of this code sequence is:

$\mathbf{C}_{split,trigger}$:

- **Input:** *trigger*.
- **Output:** *invoke*, $trigger_{E_1}, \ldots, trigger_{E_n}$.

The code sequence $\mathbf{C}_{split,send}$ splits the thread, where the pointer to the activation record is returned, into n parallel threads, one for each result of an argument expression E_i. This splitting is needed for the synchronization of the invocation of the function and the evaluation of the argument expressions, before the arguments can be sent to the function. The interface of this code sequence is therefore:

$\mathbf{C}_{split,send}$:

- **Input:** *send*.
- **Output:** $send_{E_1}, \ldots, send_{E_n}$.

$\mathbf{C}_{E_i}$, $\mathbf{C}_{SignalTree}$ are code sequences corresponding to the graphs of the expressions E_i and of the final signal tree, respectively. The input and the output of these sequences are the following labels:

$\mathbf{C}_{E_i}$, where $i = 1, \ldots, n$:

- **Input:** $trigger_{E_i}$.
- **Output:** $result_{E_i}$, $signal_{E_i}$.

$\mathbf{C}_{SignalTree}$:

- **Input:** $signal_{E_i}$, arg_{E_i}, where $i = 1, \ldots, n$.
- **Output:** *signal*.

The parallel threads that are created in $\mathbf{C}_{split,trigger}$ initiate the evaluation of the function and of the argument expressions. The next block invokes the function. The FASTCALL command performs a side-effect as it creates a dynamic link to the function f. A pointer to the activation record of the invoked function arrives at the next address after the FASTCALL command and the result of the function call arrives at the next address plus one. After the arrival of a pointer to the activation record of this function invocation, it is possible to send the arguments to the function (this is done by the code block after the corresponding argument). Therefore, this thread will be split into as many parallel threads as there are arguments for the function, which is done in $\mathbf{C}_{split,send}$. Each of these threads is synchronized with the result of the corresponding argument expression. After this synchronization the value of the argument expression is sent to the function. The last construct in this code sequence is a signal tree that joins all now superfluous threads of control and joins them to the signal output of the whole function call.

Although the translation process has only been discussed for one construct, it should be sufficient to give some feeling about how other constructs could be translated, too. For example, for translating a tuple construction one would get essentially the same structure as for a function call but with a node initiating tuple construction instead of the invoke node and nodes for writing into the fields of the tuple instead of nodes for sending arguments.

4 Conclusion

In this paper we wanted to show how a purely functional program, without any extensions for parallelism, can be translated in two steps into a program in the assembler code of a conventional control-driven machine that has two additional commands for handling parallelism. For this translation process dataflow graphs are used as the intermediate language as they show the maximal implicit parallelism of the given program. As the presented target language is based on the controlflow model, it can be implemented on conventional hardware by adding those few constructs that are necessary for parallelism.

One point that has not been discussed in this paper is the optimization of the resulting code. In the presented translation process sometimes rather inefficient code is produced. Especially, some tests with the prototype compiler have shown, that the synchronizations with the trigger token yield quite much overhead. So, one possibility for improving the efficiency of the resulting code would be to ignore the trigger token in purely arithmetical expressions since they are always strict and to forward the trigger directly to the signal token in such a case. Furthermore, a strictness analysis on user defined functions could be performed to determine the arguments in which a function is strict. When this function is called the strict arguments can then be evaluated before the function itself, thereby avoiding a synchronization with this trigger token. However, these considerations on optimizations of the resulting code are part of future work in this area.

References

[Arvind *et al.*, 1987]
Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali.
I-Structures: Data Structures for Parallel Computing.
Computation Structures Group Memo 269, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, February 1987.

Also in: Proceedings of the Workshop on Graph Reduction, Los Alamos, New Mexico, September 28 – October 1, 1986.

[Iannucci, 1988]
Robert Alan Iannucci.
A Dataflow / von Neumann Hybrid Architecture.
Technical Report MIT/LCS/TR-418, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, May 1988.

[Loogen, 1990]
Rita Loogen.
Parallele Implementierung funktionaler Programmiersprachen (Parallel Implementation of Functional Programming Languages (in German).
Number 232 in Informatik Fachberichte. Springer, 1990.

[Peyton Jones, 1989]
Simon L. Peyton Jones.
Parallel Implementations of Functional Programming Languages.
The Computer Journal, 32(2):175–186, April 1989.

[Schreiner, 1990]
Wolfgang Schreiner.
ADAM & EVE — An Abstract Dataflow Machine and Its Programming Language.
Master's thesis, Johannes Kepler University, Linz, Austria, October 1990.
Also: Technical Report 90-42.0, RISC-Linz, Johannes Kepler University, Linz, Austria, 1990. Also: Technical Report 91-1, Austrian Center for Parallel Computation, January 1991.

[Traub, 1986]
Kenneth R. Traub.
A Compiler for the MIT Tagged-Token Dataflow Architecture.
Technical Report MIT/LCS/TR-370, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, August 1986.