

An Occam Compiler for a Dataflow Multiprocessor

Bruno R. Preiss
V. Carl Hamacher

1. Introduction

In this paper we describe the implementation of an Occam language[1] compiler for a non-traditional architecture. We will describe the design of the internal representation of the Occam source program and outline some of the procedures used in the compilation process. The target processor architecture is a multiprocessor dataflow machine in which the processing elements are optimized for the evaluation of acyclic dataflow graphs[2, 3]. Consequently, the compiler is required to partition the Occam source programs into a collection of acyclic dataflow graphs.

In the target execution environment, processes are dynamically created to execute these acyclic dataflow graphs. To evaluate a dataflow graph, a process requires some input data (to be placed on the input arcs of the dataflow graph). These data are received from other processes (i.e., from the output arcs of other dataflow graphs). Consequently, the compiler is required to emit code that coordinates the communication of data between processes.

The goal in partitioning the program is the exploitation of the parallelism inherent in the program. The granularity of parallelism in this approach is controlled by the amount of computation within each element of the partition. A separate process executes each element of the partition. To emphasize the fact that these processes are comparatively small and belong to the same program, we call them *contexts*.

Choosing the size of a context, i.e. the number of nodes in the acyclic dataflow graph, involves a trade-off between the overhead of context generation and intercontext communication against the performance benefits of parallelism. If the size of contexts is decreased, then more contexts must be created and more data must be communicated between contexts. If contexts are made larger, then overall performance of the multiprocessor suffers since, in general, an individual processing element assigned to execute a context cannot exploit all intracontext parallelism.

The choice of the context as the basic granule of computation represents a compromise between the conventional dataflow execution model on the one hand, and task- or process-level parallelism of languages such as *Concurrent Euclid*[4] and *Ada*[5] on the other. The conventional dataflow execution model attempts to exploit the fine-grained parallelism at the level of individual operators and instructions. Limited success has been achieved in this approach because of the overhead associated with detecting operator-level parallelism at execution time. On the other hand, the task- or process-based approach relegates the responsibility for the detection of parallelism to the programmer. The programmer must explicitly partition a program into granules of computation. The goal in the context-based approach is to partition a program into granules of computation (contexts) which are much larger than single instructions, yet smaller than processes or tasks, and to automatically exploit the parallelism among contexts. The basic criterion for the size of a context is that the computation it contains corresponds to an acyclic dataflow

This research was supported in part by NSERC (Canada) under Grant A5192.

B. R. Preiss is with the Department of Electrical Engineering, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1.

V. C. Hamacher is with the Departments of Electrical Engineering and Computer Science and the Computer Systems Research Institute, University of Toronto, Toronto, Ontario, Canada, M5S 1A4.

graph.

The programming language chosen should lend itself as naturally as possible to the context-based paradigm for parallelism. Ideally, the separation into contexts could be based on implicit as well as explicit cues in the program itself. Languages developed for dataflow architectures such as *VAL*[6], *ID*[7], *SALAD*[8] and *LAU*[9] reflect the underlying execution mechanism. These languages are designed to permit the automatic detection of operator-level parallelism at execution time. Such languages are usually single-assignment languages and they ensure that all actors are free from side-effects and purely functional. These restrictions may tend to unduly constrain the programmer. On the other hand, languages such as *Concurrent Euclid*[4] and *Ada*[5] require the programmer to explicitly specify all the parallelism and intertask communication. Such an approach would be extremely tedious for the context-level of process granularity. In addition, such approaches usually enforce a fixed (at compile time) number of parallel processes.

2. Partitioning Occam Programs into Acyclic Dataflow Graphs

The Occam programming language[1], originally developed for the Transputer computer architecture[10], has several features which make it suitable for our context-based approach to parallelism. Occam is based on the communicating sequential process (CSP) paradigm for parallel programming. This paradigm naturally maps to contexts and intercontext communication. In addition, groups of statements in Occam must be combined into a hierarchy of sequential and parallel parts using the operators **seq** and **par**. This static hierarchy can be used to aid in the partitioning of the program into a collection of acyclic dataflow graphs. Finally, an implementation of Occam is allowed to support dynamic process creation.

In Occam there are five primitive statements, namely: assignment, input, output, wait, and skip. Each of these statements can be translated directly into an acyclic dataflow graph. Groups of statements can be combined using the **seq** and **par** constructs to form statements which can be further combined to form a hierarchy. The corresponding dataflow graphs are constructed from the dataflow graphs of the components by sequential or parallel concatenation of the component graphs.

All the preceding constructs result in a single acyclic dataflow graph. The two iteration paradigms in Occam, the **while** loop and the replicated **seq**, both would normally result in cyclic graphs. However, by using a technique analogous to recursion, it is possible to implement these constructs using acyclic dataflow graphs. This is done by using a separate acyclic dataflow graph for the loop body. Conditional execution in Occam is done with the **if** statement. A separate graph is created for the body of each branch of the **if**. The replicated **par** construct of Occam is used to dynamically create parallel processes. The implementation of this construct is similar to that of iteration — a new context is created for each instance of the **par** body. Finally, a new graph is created for each Occam subroutine (i.e., **proc** construct).

3. Dataflow Analysis and Intercontext Communication

In this section, we outline the procedures used by the compiler to construct the dataflow graph from the Occam source program. This method is based on the *flow analysis procedure for the translation of high-level languages to a dataflow graph* of Allan and Oldehoeft[11]. There are two goals in dataflow analysis. First, dataflow analysis involves the collection and organization of the information needed to construct a dataflow graph. Second, the live-value analysis phase of dataflow analysis provides optimization information which can be used to reduce intercontext communications.

In the target multiprocessor architecture, data are exchanged between contexts via hardware-assisted communication channels. These channels support the exchange of single word messages. When more than one word must be sent between two contexts, the words must be sent consecutively according to some predetermined sequence.

Each context corresponds to an acyclic dataflow graph. Thus, the messages that are received by a context correspond to the input arcs of the dataflow graph. The messages sent by a context correspond to the output arcs of the dataflow graph. In the second subsection below we describe a method for sequencing the input arcs of a dataflow graph. This sequencing is used to guide the compiler in emitting instructions for intercontext communication.

3.1. Analysis

The first step in the analysis procedure is the construction of a representation of the Occam source program based on the use of an intermediate form table (IFT). Each entry of the IFT has five fields:

1. the type of the entry,
2. I , the set of input values for the entry,
3. O , the set of output values for the entry,
4. T , the syntax tree associated with the entry, and
5. $E = \{E_1, E_2, \dots, E_n\}$, an ordered set, E , each of whose elements E_i is an ordered set of IFT indices.

There are two kinds of IFT entry. *Non-interface* IFT entries correspond to simple high-level entities, i.e. Occam primitives, conditions, and replicators. These entries have an empty E set and may have non-empty syntax trees. The rules for constructing non-interface IFT entries are shown in Table I.

The *interface* IFT entries correspond to compound high-level statements and may have non-empty E sets, but always have empty syntax trees. An interface entry represents a staging area for values used within and produced by a block of instructions. The rules for constructing the interface IFT entries for Occam programming language constructs are shown in Table II.

The purpose of the E set is to represent the hierarchical nature of the Occam program. Its elements correspond to the components of the construct that may proceed in parallel. Each of these components is a set of operations that must be done sequentially.

There are three steps in the dataflow analysis procedure. First, the IFT is constructed during the parsing of the Occam program and the I , O , and E sets are constructed as described above. Second, the I and O sets are modified by associating two sets of IFT indices with each value. These sets, D and U , establish a relationship between the

definition and subsequent use of a value.

For each element x of the I set of an entry, its D set contains the index of the IFT entry where the value is defined. For each element x of a *non-interface* entry, the U set is empty. For each element x of an *interface* entry, the U set contains the IFT entry indices of the places where the value is used *within* the scope of the entry.

Conversely, for each element x of the O set of an entry, its U set contains the index of the IFT entry where the value is used. For each element x of a *non-interface* entry, the D set is empty. For each element x of an *interface* entry, the D set contains the IFT entry indices of the places where the value is defined *within* the scope of the entry.

The definitions of U and D presented here are a generalization of those used by Allan and Oldehoeft. Our method differs from theirs in that the elements of our O set have both a U and D set associated with them, whereas they only use the U set.

The third and final step in the dataflow analysis procedure is live-value analysis, which associates with each value in every O set a Boolean-valued tag which indicates whether the value has a subsequent use in the program.

The U and D sets are constructed using the *UseAndDef* procedure (Algorithm 1). The *UseAndDef* procedure is a top-down recursive descent algorithm which calls *FindDef* to link the elements of the I and O sets of IFT entries. The *FindDef* procedure scans through the ordered list of IFT indices P (corresponding to statements preceding H_j in the current scope) searching for an entry whose output set contains the value x . If it is not found in P , then *FindDef* checks the input set of H , the interface entry for this scope, to see if the value is imported from the enclosing scope. If the definition is found, the appropriate U and D sets are modified to reflect the dependency.

The *LiveAnalyze* procedure (Algorithm 2), performs the live-value analysis. The rules for determining whether a value is live are as follows:

1. A value which has a non-empty U set which is not equal to $\{H\}$, the containing interface IFT entry, is live.
2. A value whose U set is $\{H\}$ and the entry type of entry H is a loop (i.e. **while** or replicated **seq**) and the value is in the input set of the loop is live. Otherwise, the liveness of the value is obtained from the enclosing scope.
3. All **var** formal procedure arguments are live.

Once the IFT has been constructed and dataflow analysis completed, the dataflow graph can be easily constructed. The augmented IFT itself contains a representation of the dataflow graphs corresponding to the program. The process of constructing the actual graphs is a simple, but tedious job of translating the IFT to a suitable representation.

3.2. Intercontext Communication

The optimum sequence of values for intercontext communication is that sequence which results in the minimum total execution time for a given program. Unfortunately, this definition is unworkable because it requires that a global optimization be performed on all the graphs associated with a given program. Since a program is partitioned into many small graphs, each of which transmits and receives many values, the search space of possible value sequences becomes large. Instead, the following heuristic assumption is made: *The preferred sequence of input values for intercontext communication is that sequence which maximizes the amount of computation possible in a*

context before the context must wait for another input. This definition allows us to consider only the input arcs of a dataflow graph. On the basis of this heuristic, we define a partial order relation on the inputs of an acyclic dataflow graph. This partial order relation describes the sequencing constraints on the inputs to the graph.

Definition: The set of immediate predecessors of vertex $v \in V$ of an acyclic dataflow graph $G=(V,E)$ is the set $P(v) \subseteq V$ given by $P(v)=\{u \in V: (u,v) \in E\}$.

Definition: The set of immediate successors of vertex $v \in V$ of an acyclic dataflow $G=(V,E)$ is the set $S(v) \subseteq V$ given by $S(v)=\{w \in V: (v,w) \in E\}$.

Definition: The predecessor set $P^*(v)$ of a vertex v of an acyclic dataflow graph $G=(V,E)$ is the set of all the nodes in G preceding v and including v itself, i.e.

$$P^*(v)=\{v\} \cup \bigcup_{u \in P(v)} P^*(u).$$

Definition: $C(v)$ is the cost of the computation associated with a node v of an acyclic dataflow graph $G=(V,E)$.

Definition: $C^*(v)$ is the cumulative cost of the computation associated with a node v of an acyclic dataflow graph $G=(V,E)$ given by $C^*(v)=\sum_{v' \in P^*(v)} C(v')$.

Definition: The required input set $I^*(v)$ of a node v of an acyclic dataflow graph $G=(V,E)$ is the set $I^*(v)=P^*(v) \cap I$.

The preceding definitions associate a cost of computation and the inputs required for that computation with every node of an acyclic dataflow graph. Using these concepts, it is now possible to define a relation on the set of inputs I of an acyclic dataflow graph.

Definition: π_I is a relation defined on the set of input nodes of an acyclic dataflow graph $G=(V,E)$ given by:

$$\forall a,b \in I: a \pi_I b \Leftrightarrow (a=b) \text{ or } ((a \neq b) \text{ and } (\sum_{v \in V: a \in I^*(v)} C^*(v) > \sum_{w \in V: b \in I^*(w)} C^*(w))).$$

This relation is a partial order[2].

Definition: A sequence $\{v_1, v_2, \dots, v_{|V|}\}$ of the inputs I of an acyclic dataflow graph $G=(V,E)$ satisfies the relation π_I if and only if $\forall i, j, 1 \leq i < j \leq |V|: \neg (v_j \pi_I v_i)$.

We now describe an algorithm for constructing a sequencing of the inputs to an acyclic dataflow graph satisfying the π_I relation. The first step of the algorithm is to construct a depth-first list $L=\{l_1, l_2, \dots, l_{|V|}\}$ of the vertices of the graph $G=(V,E)$ (Algorithm 3). A depth-first list has the property that all the successors of a node precede the node in the list.

The second step of the algorithm is to compute for each vertex $v \in V$ of the dataflow graph the sets $P^*(v)$ and $I^*(v)$ and the cumulative computational cost $C^*(v)$. This is easily done using the depth-first list of nodes (Algorithm 4).

The final step of the algorithm involves computing the values $W(v)=\sum_{u \in V: v \in I^*(u)} C^*(u)$ for each of the inputs v to the dataflow graph (Algorithm 4). The desired input arc sequence is obtained by sorting the input arcs of the dataflow graph according to W .

4. Conclusions

We have described the implementation of an Occam compiler for a multiprocessor dataflow machine. We showed how Occam programs are partitioned into acyclic dataflow graphs. We have adapted the dataflow analysis and live-value analysis procedures of Allan and Oldehoeft to Occam. In addition, we presented an algorithm for choosing a sequencing of the input arcs for a dataflow graph. This sequence is used to specify intercontext communications. This algorithm is based on the heuristic assumption that total execution time will be minimized when the amount of computation within a context is maximized before (another) input is required. Preliminary empirical results have shown that this heuristic can reduce total execution time by up to 10 percent.

5. References

1. INMOS Ltd., *OCCAM Programming Manual*, Englewood Cliffs, NJ: Prentice-Hall, 1984.
2. B. R. Preiss, "Data Flow on a Queue Machine," Ph.D. Thesis, Dept. Elec. Engin., Univ. of Toronto, 1987.
3. B. R. Preiss and V. C. Hamacher, "Data Flow on a Queue Machine," *Proc. 12th Ann. Symp. Comput. Arch.*, June 1985. pp. 342-351,
4. R. C. Holt, *Concurrent Euclid, the Unix System, and Tunis*, Reading, MA: Addison-Wesley, 1983.
5. DOD, "Reference Manual for the ADA Programming Language," ANSI/MIL-STD-1815A-1983, U.S. Dept. of Defense, Washington, DC, 1983.
6. J. B. Dennis, G.-R. Gao, and K. W. Todd, "Modeling the Weather with a Data Flow Supercomputer," *IEEE Trans. Comput.*, Vol. C-33, No. 7, July 1984. pp. 592-603,
7. Arvind, K. P. Gostelow, and W. Plouffe, "An Asynchronous Programming Language and Computing Machine," Tech. Rep. TR 114a, Dept. Inf. and Comput. Sci., UCI, Dec. 1978.
8. T. Christopher, O. El-Dessouki, M. Evans, H. Harr, H. Klawans, P. Krystosek, R. Mirchandani, and Y. Tarhan, "SALAD — A Distributed Compiler for Distributed Systems," *Proc. 1981 Int. Conf. Parallel Processing*, Aug. 1981. pp. 50-57,
9. A. Plas, "LAU System Architecture: A Parallel Data-Driven Processor Based on Single Assignment," *Proc. 1976 Int. Conf. Parallel Processing*, Aug. 1976. pp. 293-302,
10. C. Whitby-Stevens, "The Transputer," *Proc. 12th Ann. Symp. Comput. Arch.*, June 1985. pp. 292-300,
11. S. J. Allan and A. E. Oldehoeft, "A Flow Analysis Procedure for the Translation of High-Level Languages to a Data Flow Language," *IEEE Trans. Comput.*, Vol. C-29, No. 9, Sept. 1980. pp. 826-831,

```
procedure UseAndDef ( $H$ : IFTindex)
  local var  $P$ : ordered list of IFTindex

  for each  $E_i \in E(H)$ 
     $P \leftarrow \emptyset$ 
    for each  $H_j \in E_i$ 
      for each  $(x, D, U) \in I(H_j)$ 
        FindDef ( $x, H_j, H, P, D$ )
      end for each
      UseAndDef ( $H_j$ )
       $P \leftarrow$  concatenate ( $\{H_j\}, P$ )
    end for each
    for each  $(x, D, U) \in O(H)$ 
      FindDef ( $x, H, H, P, D$ )
    end for each
  end for each
end procedure UseAndDef

procedure FindDef ( $x$ : symbol,  $H_j, H$ : IFTindex,
   $P$ : ordered list of IFTindex,  $D$ : set of IFTindex)
  for each  $H_k \in P$ 
    if  $(x, D', U') \in O(H_k)$  then
       $U' \leftarrow U' \cup \{H_j\}$ 
       $D \leftarrow D \cup \{H_k\}$ 
    return
    end if
  end for each
  if  $(x, D', U') \in I(H)$  then
     $U' \leftarrow U' \cup \{H_j\}$ 
     $D \leftarrow D \cup \{H\}$ 
  end if
end procedure FindDef
```

Algorithm 1.

```

procedure LiveAnalyze ( $H$ : IFTindex)
  for each  $E_i \in E(H)$ 
    for each  $H_j \in E_i$ 
      for each  $(x, D, U, Live) \in O(H_j)$ 
        if  $U \neq \emptyset$  then
          if  $type(H)$  is a loop and
             $x \in I(H)$  then
               $Live \leftarrow \text{true}$ 
            else
               $find (x, D', U', Live') \in O(H)$ 
               $Live \leftarrow Live'$ 
            end if
          else
            if  $x$  is a var formal then
               $Live \leftarrow \text{true}$ 
            else
               $Live \leftarrow \text{false}$ 
            end if
          end if
        end for each
      end for each
    end for each
  end procedure LiveAnalyze

```

Algorithm 2.

```

procedure ConstructDepthFirstList
  initially all nodes  $v \in V$  are unmarked
   $i \leftarrow 1$ 
  loop
    exit when all nodes  $v \in V$  are marked
    choose an unmarked  $v \in V$ 
    Search ( $v$ )
  end loop
end procedure ConstructDepthFirstList

procedure Search ( $n$ : node)
  mark  $n$ 
  for each  $m \in S(n)$ 
    if  $m$  is not marked then
      Search ( $m$ )
    end if
  end for each
   $l_i \leftarrow n$ 
   $i \leftarrow i+1$ 
end procedure Search

```

Algorithm 3.

```

for  $i: |V| \cdots 1$ 
   $P^*(l_i) \leftarrow \{l_i\}$ 
  if  $l_i \in I$  then
     $I^*(l_i) \leftarrow \{l_i\}$ 
  else
     $I^*(l_i) \leftarrow \emptyset$ 
  end if
  for each  $m \in P(l_i)$ 
     $P^*(l_i) \leftarrow P^*(l_i) \cup P^*(m)$ 
     $I^*(l_i) \leftarrow I^*(l_i) \cup I^*(m)$ 
  end for each
   $C^*(l_i) \leftarrow \sum_{l_j \in P^*(l_i)} C(l_j)$ 
end for

```

Algorithm 4.

```

for each  $v \in I$ 
   $W(v) \leftarrow 0$ 
  for each  $u \in V$ 
    if  $v \in I^*(u)$  then
       $W(v) \leftarrow W(v) + C^*(u)$ 
    end if
  end for each
end for each
sort the elements of I according to W

```

Algorithm 5.

TABLE I
NON-INTERFACE IFT ENTRIES

type	example	I	O	E	T
assignment	$z := x + y$	$\{x, y\}$	$\{z\}$	\emptyset	$(:=, z, (+, x, y))$
input	$c ? x$	$\{K, c\}$	$\{K, x\}$	\emptyset	$(?, c, x)$
output	$c ! x + y$	$\{K, c, x, y\}$	$\{K\}$	\emptyset	$(!, c, (+, x, y))$
wait	wait now after x	$\{K, x\}$	$\{K\}$	\emptyset	(wait, x)
skip	skip	\emptyset	\emptyset	\emptyset	$()$
condition	$x < y$	$\{x, y\}$	\emptyset	\emptyset	$(<, x, y)$
replicator	$i = [x \text{ for } y]$	$\{x, y\}$	$\{i\}$	\emptyset	(rep, i, x, y)

TABLE II
INTERFACE IFT ENTRIES

construct	I, O, E
seq P_1 P_2 \dots P_n	$I = I(P_1) \cup \left\{ \bigcup_{i=2}^n (I(P_i) - \bigcup_{j=1}^{i-1} O(P_j)) \right\}$ $O = \bigcup_{i=1}^n O(P_i)$ $E = \{\{\rho_1, \rho_2, \dots, \rho_n\}\}$ where ρ_i is the IFT index of P_i .
par P_1 P_2 \dots P_n	$I = \bigcup_{i=1}^n I(P_i)$ $O = \bigcup_{i=1}^n O(P_i)$ $E = \{\{\rho_1\}, \{\rho_2\}, \dots, \{\rho_n\}\}$
if C_1 P_1 C_2 P_2 \dots C_n P_n	$I = \bigcup_{i=1}^n \{I(C_i) \cup (I(P_i) - O(C_i))\}$ $O = \bigcup_{i=1}^n (O(C_i) \cup O(P_i))$ $E = \{\{\gamma_1, \rho_1\}, \{\gamma_2, \rho_2\}, \dots, \{\gamma_n, \rho_n\}\}$ where γ_i is the IFT index of C_i .
while C_1 P_1	$I = I(C_1) \cup (I(P_1) - O(C_1))$ $O = O(C_1) \cup O(P_1)$ $E = \{\{\gamma_1, \rho_1\}\}$
seq R_1 P_1	$I = I(R_1) \cup (I(P_1) - O(R_1))$ $O = O(P_1)$ $E = \{\{r_1, \rho_1\}\}$ where r_1 is the IFT index of R_1 .
par R_1 P_1	$I = I(R_1) \cup (I(P_1) - O(R_1))$ $O = O(P_1)$ $E = \{\{r_1, \rho_1\}\}$