A DEDICATED DATA FLOW ARCHITECTURE FOR HARDWARE COMPILATION

Majid Mojtabavi Naini

Department of Electrical and Computer Engineering
Florida Atlantic University
Boca Raton, Florida 33431

ABSTRACT

This paper describes a dedicated data flow architecture which has been designed to be a part of the HARDWARE COMPILER. This machine evaluates attribute grammars in a data flow fashion by accepting their reverse dependency graph which is similar to a data flow graph. The outputs and the results of these evaluations are sent to the other components for later use. This machine appears to be the first dedicated data flow architecture suggested for this purpose. The machine takes advantage of parallelism at two levels; first, the components of the machine are organized in a pipeline fashion and can run concurrently; second, the execution of the instructions are done in parallel as well.

1. INTRODUCTION

The ultimate goal in designing any real time computer system should be higher speed, more efficiency, and better economics. Speed is a very significant issue for all computers and especially for the real time systems. Efficiency is important in both software and hardware design. From the software point of view the programmer should be able to construct his programs in parts, modify them, correct the errors as they are detected, and have a new compiled version of his changed program instantly [2]. The recompilation of a modified program is time consuming and, as a result, expensive. To improve the computation efforts and the system response time, compilation can be put at the hardware level rather than the software level. From the hardware end, the rapid advancement in VLSI design suggest the decreasing

hardware costs and greater reliability in the very near future. Computer architects may think seriously of designing new dedicated architecture for different levels of a computer system. Some important advantages of this outlook are speed, reliability, and low cost. The HARDWARE COMPILER is an example of this new way of thinking.

A central theme of computer architecture is that "hardware and software are logically equivalent" (Tanenbaum)[16]. Any operation can either be implemented by software or directly built into the hardware. Cost, speed, reliability, and frequency of expected changes are the main factors effecting the decision of which functions should be built into hardware and which should be programmed explicitly. The boundary between software and hardware is arbitrary and continually changing and "today’s software is tomorrow’s hardware" (Tanenbaum, [16]). Moreover, the different levels of the machine have also flexible boundaries. The user is concerned with speed, rather than how the instructions are implemented.

Traditionally compilers have been implemented by software programs where the concerns are syntax and the semantics [1]. Attribute grammars are one of the best formalisms to define the semantics of the programming languages [9]. Several efficient software compilers have been developed which use attribute grammars [6,7,8]. Attribute grammars can be used to specify the translation and code generation by allowing the attributes to represent data types of expressions, symbol tables, machine code, etc [10]. Using attribute
grammars to define a programming language and its computation process offers several advantages: (1) The semantics is descriptive and independent of any parsing method; (2) The semantic description is modular and is given on a production by production basis; (3) Semantic understanding and language modifications are easily expressed; (4) Non context-free features are easily expressed.

This paper describes a data flow machine which is a component of the hardware compiler. The machine receives a graph called reverse dependency graph which is very similar to a data flow graph and evaluates this graph in a data flow manner which results in parallel evaluation of the attributes. This machine has a multi-parallel architecture. The segments of the machine are organized in a pipeline manner so can run concurrently and the execution of the instructions are done in parallel too.

The machine presented here in conjunction with the PARSER MACHINE and other necessary components will be built on VLSI chips. The compiler will also be an incremental one and as a result of the above two factors, it will be much faster, more economical, and more efficient. The VLSI chips could replace the traditional compilers for the real time systems. The structure of this compiler is also different and more efficient because of the use of attribute grammars and also because is an incremental compiler.

This paper is organized as follows. Section 2 reviews attribute grammars. In section 3 the attribute evaluation is described. Section 4 explains the architecture of the machine. In Section 5 the operation of the machine is described. Finally Section 6 contains the concluding remarks about the system. The future research and the extensions of the work presented here are addressed in section 7.

2. REVIEW OF THE ATTRIBUTE GRAMMARS

Knuth [9] proposed attribute grammars to specify the semantics of programming language whose syntax is defined by context free grammars [3,4]. An attribute grammar is an ordinary context-free grammar extended to specify the "meaning" of each string in the language. Each grammar symbol has a set of attributes associated with it. These attributes are defined in terms of other attributes and each production rule is provided with corresponding semantic rules expressing the relationship between the attributes of the symbol in the production rule. These attributes take their values from some given domains. The purpose of these attributes is to transmit information from nodes to nodes in the parse tree for a given input. There are two kinds of attributes: (1) Synthesized attributes transmit information from bottom-up; (2) Inherited attributes transmit information from top-down.

The meaning of an input string is the value of the attributes of the start symbol in the parse tree for that input. This is determined by semantic evaluation of its parse tree. Given a parse tree $\mathcal{T}$ for an input string $w$, a system $E$ of equations is obtained from the semantic rules associated with the productions involved in deriving the parse tree. The least solution of this system is then computed and the components of this solution, corresponding to the synthesized attributes of the root of the tree, are taken as the semantics of the input string. An algebraic parse tree is a parse tree whose nodes are labelled by the productions instead of grammar symbols. A semantic tree is an algebraic parse tree which is augmented by attaching to every node a list of fields used for storing the values of attribute instances. Additional list of references on attribute grammar can be found in [11,13,14,15].

3. ATTRIBUTE EVALUATION

Attribute evaluation of a semantic tree $\mathcal{T}$ is complete when all the synthesized attribute instances of the root are evaluated. To evaluate an attribute instance $a(u)$ (synthesized or inherited), we must first evaluate the attribute instances which are the arguments of the semantic function defining it. Hence, evaluation of the synthesized attributes of the root leads to a recursive evaluation of the attributes.
For every production $P: X_0 \rightarrow X_1 X_2 \ldots X_n$, the semantic dependencies of the attribute occurrences of $P$ can be described by $DG_P$, the DEPENDENCY GRAPH of production $P$. The nodes of $DG_P$ are the attribute occurrences of $P$. There exists an edge from node $n_i$ to node $n_j$ if $n_j$ is an argument of the semantic function defining $n_i$. The semantic dependencies of a semantic tree $T$ can be represented by the graph $DG_T$ called the COMPOUND DEPENDENCY GRAPH of $T$. The graph $DG_T$ is constructed by attaching together copies of the $DG_P$'s for the productions occurring in the tree.

EXAMPLE ([9]).

<table>
<thead>
<tr>
<th>Production Rules</th>
<th>Semantic Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_1: B \rightarrow 0$</td>
<td>$B.v(0) = 0$</td>
</tr>
<tr>
<td>$p_2: B \rightarrow 1$</td>
<td>$B.v(0) = 2^{v(s)}$</td>
</tr>
<tr>
<td>$p_3: L \rightarrow B$</td>
<td>$L.v(0) = B.v(1)$</td>
</tr>
<tr>
<td></td>
<td>$L.l(0) = 1$</td>
</tr>
<tr>
<td>$p_4: L \rightarrow LB$</td>
<td>$L.v(0) = L.v(1) + B.v(2)$</td>
</tr>
<tr>
<td></td>
<td>$L.l(0) = L.l(1) + 1$</td>
</tr>
<tr>
<td>$p_5: N \rightarrow L$</td>
<td>$N.v(0) = L.v(1)$</td>
</tr>
<tr>
<td>$p_6: N \rightarrow LL$</td>
<td>$N.v(0) = L.v(1) + L.v(3)$</td>
</tr>
</tbody>
</table>

The attributes $v$, $l$, and $s$ stand for value, length and scale. (The slightly redundant notation for attribute instances is used to help the reader's understanding, i.e. $L.v(3)$ for $v(3)$).

Dependency Graphs of the Production Rules

- $DG_1: B.v(e) \quad B.s(e)$
- $DG_2: B.v(e) \quad B.s(e)$
- $DG_3: B.v(1) \quad B.s(1)$
- $DG_4: L.v(1) \quad L.l(1) \quad L.s(e)$
- $DG_5: L.v(1) \quad L.l(1) \quad L.s(1) \quad B.v(2) \quad B.s(2)$
- $DG_6: N.v(e)$
- $DG_7: L.v(1) \quad L.l(1) \quad L.s(1)$
- $DG_8: L.v(1) \quad L.l(1) \quad L.s(1) \quad L.v(3) \quad L.l(3) \quad L.s(3)$
Algebraic parse tree for input string 10.1:

```
       p3
      /  \
     p4   p4
    /  \
   p1   p2
```

Dependency graph DG for the algebraic parse tree for input 10.1

In order to evaluate $N.v(e)$, the following subtree of the dependency graph DG is visited:

The value of $N.v(e)$ is 2.5. Only the attribute instances reachable from $N.v(e)$ need to be evaluated. Hence, attribute instances $L.l(1)$, $L.l(11)$ and $B.s(12)$ need not be evaluated.

### 3.1 DATA FLOW EVALUATION OF THE ATTRIBUTES

This evaluation is very similar to the data flow evaluation [12]. The reverse compound dependency graph plays the same role for the attribute evaluation as the data flow graph does for the data flow evaluation [5]. The evaluation concept is the same in both cases. Like the data flow evaluation, the attribute evaluation takes place when all the elements necessary for the evaluation are available. The reverse...
compound dependency graph shows the order in which the attributes are evaluated. The leaf attributes initially all have values and give their values to their predecessor attributes so they can be evaluated as well. In general, each node in the graph is evaluated when all of its successors have values. Since the evaluation takes place according to the order of the attributes in the reverse compound dependency graph from bottom-up, all of the predecessor attributes of the leaves which have values are candidates for the evaluation.

4. ARCHITECTURE OF THE MACHINE

This machine has a parallel architecture and the parallelism is exploited at two levels. First, the machine components are organized in a pipeline fashion in which they can run concurrently. Second, the execution of the instructions are done in parallel as well. Figure 1. shows the different components of the machine. These components are PROCESSING UNIT, UPDATE CONTROLLER, MEMORY, ATTRIBUTE POOL, and SWITCH. The memory itself consists of three parts: GRAPH memory, SEMANTIC MODULES memory, and DYNAMIC MAP memory. Each memory has its own memory buffer register.

One of the advantages of having three separate memories is the fact that the operations of these memories could be overlapped. The update controller can access all of these memories simultaneously and perform a read or write operation; therefore, the access to the information stored in these memories is almost three times faster than if it was stored in one memory.

Figure 1. Machine Organization
4.1 PROCESSING UNIT

The processing unit has several processing elements which can run simultaneously and in parallel. Each processing element could be a general purpose processor or a special purpose processor. The processing unit receives the attribute packages from the attribute pool, executes them in parallel, and sends the results to the update controller.

4.2 MEMORY

The memory consists of three different parts as follows:

1) Graph memory is a random access memory and contains the reverse compound dependency graph. This graph as was mentioned earlier shows the attribute dependency and the order of their evaluation. The structure of the graph memory is depicted in Figure 2. Each graph memory word has several fields. The attribute field contains the attribute code for the attribute instance in the graph. The predecessor address field contains the addresses of the attribute predecessor in the static dependency graph. The first bit of the predecessor address field is called the validity bit which indicates if this address is valid or not. The predecessor address is the address of a location in the graph memory. The successor address fields contain the addresses of the attributes succeeding this attribute. Again these addresses indicate locations in the graph memory and the first bit of each successor address is called the validity bit with the same function as above. The last field holds the address of the dynamic bit memory for the attribute instance.

2) The semantic module memory is a random access memory in which the semantic modules reside. These semantic modules are the semantic rules which are precompiled into the routines. These routines are executed by the processors to compute the values of the attributes.

3) The dynamic map memory is shown in Figure 3. Each row of the dynamic map memory indicates the dependency of an attribute on the other attributes and it contains several fields, some of them the same nature. These fields are validity bit, attribute code, attribute value, and an address of the semantic module memory. The validity bit indicates the availability of an attribute which is represented by attribute code and attribute value. The reason for their existence will become clear later.

4.3 UPDATE CONTROLLER

The update controller is in charge of sending information back and forth in the machine. It accesses the graph memory, dynamic map memory, and the semantic modules memory and can update their information. It forms the attribute packets and sends them to the attribute pool which in turn sends these packets to the processing unit. An attribute packet consists of the attribute values plus the semantic modules of that attribute which is ready for evaluation and the destination address. The semantic modules with the values will be used by the processors to calculate new values. The destination address is the address of the attribute which is receiving this new value in the graph memory. The controller also receives the results from the processing unit and sends them to the related component.

The update controller has a series of registers which keep the information temporarily before it is sent to the pertinent components. It has its own service processors that provide the required operations during the preoperational and operational phases. The microprograms controlling these operations are written on PROMS and are contained in the update controller. The outline of these microprograms are presented in the following sections.

4.4 ATTRIBUTE POOL

The attribute pool is a random access memory with multiaccess which contains attribute packets. The attribute pool is used to regulate the traffic in the pipeline ring. It receives the attribute packets and sends them to the processing elements for the execution.

4.5 SWITCH

The switch connects the machine to the outside world. The switch receives the reverse dependency graph as an input and passes it on to the update controller. It also receives the result of the attribute evaluation from the update controller and sends them out as output. This output is used by other segments of the hardware compiler.

5. OPERATION OF THE SYSTEM

The system's operation is divided into two phases. Phase one is called the preoperational phase and phase two is called the operational phase.
5.1 PREOPERATIONAL PHASE

In the preoperational phase the update controller receives the reverse dependency graph from the switch and loads the graph memory with this graph. The loading process takes place in the following manner. Each attribute instance is assigned a code number, predecessor addresses, successor addresses, and a dynamic map memory address. Each row of the graph memory is assigned to an attribute instance, and each field of the row is loaded with pertinent information. The loading of the dynamic map memory also takes place in this phase. The attribute instances present in the reverse dependency graph would be copied into the dynamic map memory in the following order. Each row of the dynamic map memory would be assigned to an attribute instance and the attribute instances present in the dynamic map memory would be also loaded in the preoperational phase by putting the attribute codes and values which are necessary for the evaluation of that attribute would be loaded into that row. As was shown before in Figure 3, each row of the dynamic map memory has been divided into several fields and each of these fields represents an attribute instance necessary for the evaluation of the attribute instance represented by this row. Each field has some subfields. The subfield called validity bit would be set during this loading process if the value of its attribute is ready at this time. This would be known after the construction of the dependency graph. Therefore, each row of a dynamic map memory represents an attribute and its dependency on the other attributes for its evaluation and the validity bit indicates the presence of the attribute value. The semantic module memory would also be loaded in the preoperational phase by putting the
semantic modules in this memory. The semantic module address field of the dynamic map memory is used to find these locations in the semantic module memory and load the pertinent semantic modules in these locations. The update controller forms the attribute packets from the known attribute and puts them in the attribute pool for the execution. A microprogram is given below to better explain this process. It should be noted that each instruction is a macro instruction which is not explained further here for lack of space. The execution of these instructions could be overlapped to achieve a better speed.

**MICROPROGRAM OF THE PROOPERATIONAL PHASE**

```
LOAD graph memory
LOAD dynamic map memory
LOAD semantic module memory
FORM attrib-packet
LOAD attribute pool
```

**5.2 OPERATIONAL PHASE**

In this phase the actual evaluation of the attributes takes place by execution of the semantic modules by the processor elements. The processing unit removes an attribute packet from the attribute pool and sends it to one of the processing elements for the execution. The update controller receives the result from the processing unit and puts it into one of its registers and uses the destination address of the result to access the graph memory. This row of the graph memory has been assigned to the attribute whose value has just arrived, so the attribute code of this attribute (from the attribute code field) is put in one of the controller registers to be used later in updating the dynamic map memory. Hence the graph memory is updated to reflect the changes in the attribute codes of the predecessor attributes. These addresses are kept in controller registers and are used to access the graph memory. The graph register indicates the attributes which are necessary for the evaluation of the predecessor attributes and the dynamic map register indicates the presence or the absence of these attributes. By comparing these registers it can be determined if the predecessor attributes are ready for evaluation or not. If they are, the update controller gets the attribute values from dynamic map memory and packages them with the semantic modules of the predecessor attributes to form the attribute packets. The semantic modules are retrieved from the semantic module memory by accessing the respective locations of the memory which holds the semantic modules for the predecessor attributes. At last, these packets are formed and put in the attribute pool and the process will continue. A microprogram in the next page illustrates this phase in more detail. It should be noted that some of the instructions are macro instructions which are not explained here because of space limitation.

**6. CONCLUDING REMARKS**

The machine presented here has decentralized control. The update controller is in charge of transferring information among memories, switch, and the attribute pool. The processing unit is responsible for getting attribute packets from the attribute pool, executing them, and sending the results back to the update controller. This method avoids a heavy burden on the update controller and prevents a bottleneck in the system. It should be noted that the speed limitation of the machine is directly related to the access time of the memories, since the number of processing elements could be increased very easily.

This machine appears to be the first machine suggested and designed for this purpose and it is a new dedicated multiparallel data flow architecture. Some important advantages of hardware compilations are higher speed, more dependability and lower cost in the near future. The proposed hardware compiler will be a great hardware component for the real-time computers.

**7. FUTURE EXTENSIONS OF THIS RESEARCH**

The continuation of this research will be the subject of future publications. They will include the completion of the PARSER MACHINE design, and the implementation of the HARDWARE COMPILER. There is no PARSER MACHINE in existence at present time. This machine will have a parallel architecture and in conjunction with the attribute machine and other necessary component will be built on custom made VLSI chips to make a very powerful HARDWARE COMPILER.
MICROPROGRAM OF THE OPERATIONAL PHASE

BEGIN

FETCH attrib-packet  (An attribute-packet is removed from the attribute pool and is sent to the processors to be executed.)

EXECUTE attrib-packet

GET attrib-result  (Receive attribute-result from the processing unit and keep it in one of the controller registers.)

ACCESS graph memory  (Destination address of the attrib-result is used to find this location in the graph memory.)

FIND predecessor-attrib  (Predecessor address field is used to find this attribute predecessor in the graph memory. This address and the attribute code of predecessor attribute are kept in some controller registers. The address is used to form attribute packet and the attribute code is used in later stages.)

FOR each predecessor-attrib DO

GET successors-attrib  (Successor addresses fields in pre-attrib are used to find these attributes in the graph memory.)

LOAD G-register  (The attribute code of successor attributes are loaded into the graph register.)

ACCESS dynamic map memory  (The address of this memory is obtained from the DM MEM address field of the predecessor attribute address field.)

UPDATE dynamic map memory  (The value of attribute-result is placed in the value field of the accessed location.)

LOAD DM-register  (Dynamic map register is loaded with the attribute codes which their validity bit is set.)

MASK SG-register WITH DM-register  (By comparing the content of these registers which are the attribute codes the readiness of the attribute for evaluation is determined.)

IF MASK=1 THEN ACCESS semantic module memory  (The semantic module address field of the dynamic map memory is used to access this location in the semantic module memory.)

GET semantic module (pre-attrib)  (The semantic module necessary for forming the attribute-packet is obtained.)

GET attrib-values  (The attribute values are obtained from the DM memory to form the attribute-packet.)

FORM attrib-packet

LOAD attrib-packet

ELSE start again

END
REFERENCES


