IEEE Region 10 Conference, Tencon 92
11th - 13th November, 1992
Melbourne, Australia

ARDEN ----- ARchitecture Development ENvironment

Feipei Lai, Fangwui Tsaur, Rung-Ji Shang
Dept. of Computer Science
and Information Engineering
& Dept. of Electrical Engineering
National Taiwan University
Taipei, Taiwan, R.O.C
Email : Flai@cad.ee.ntu.edu.tw

Abstract

Software supports, such as compiler and simulation tools, are increasingly crucial for architecture development. A flexible system called *Arden* is being developed to help evolve efficient architecture. *Arden* combines a retargetable compiler with a back-end simulation tool which provides quantitative information for making a good decision in architecture designs. User-oriented specification for the code generator has simplified the machine descriptions and uses only 31 rules in describing DLX architecture. An experimental bottom-up matching algorithm which reduces the pattern matching to a numerical computation problem and improves the space complexity is also presented here.

1. Introduction

Quantitative approach in architecture has been successful for providing decisive information during architecture designs. A flexible compiler and adaptive simulation tools are the prerequisite for such an approach. There are some retargetable compilers [1, 2, 3, 4] which have attempted to improve the speed of generated code-generator and the quality of output code in order to compete with a hand-crafted code generator for specific machines. But, to be a retargetable compiler, there should be another critical component that is the specification facilitation of machine-dependent ingredient. The major goals of Arden are to reduce the complexity of describing architecture characteristics, to eliminate the gap between the compiler and the architecture designer, and to integrate the compiler with the simulator into a flexible environment.

Some of the developed code generator generators have their defects at targeting to one specific machine. For example:

1) Retargetable compiler GNU cc [11] is being updated and redistributed periodically. However, our experience with GNU shows that it is hard to write the instruction descriptions with peephole optimization for a specific instruction set. On the other hand, GNU cc adopts ***intermediate pattern oriented*** instead of ***instruction oriented*** instruction descriptions such that users must check all the patterns in the description file while one instruction is added to or deleted from the instruction set.
2) Twig[1] and BEG[4] used dynamic programming to guarantee that the code generator will output instructions with minimum cost for each subject tree pattern. They associated cost variation with each instruction pattern instead of defining peephole optimization pattern in instruction descriptions. But, the writers of instruction description must take care of the completeness of description by themselves. The code generator in *Arden* uses dynamic programming technology to get optimal code as Twig did and adds semantic-equivalent translation scheme to pattern matching phase for facilitating instructions description. The semantic-equivalent translator translates an unmatched pattern into the pattern defined in instruction descriptions.

Pattern matching usually dominates the speed of retargetable code generator (about 80 percent of the time in twig-generated code generator is devoted to the pattern matching [1]). Section 4 gives the details of pattern matching algorithm.

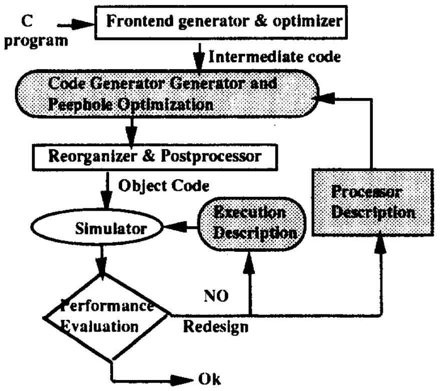

Fig. 1 Architecture Development Environment

Figure 1 presents a rough overview of *Arden* and the details of critical components which are shadowed in

0-7803-0849-2/92 $3.00 © 1992 IEEE

Figure 1 will be discussed separately. The following section will give an overview of the prototype compiler system in *Arden* and the description language of instruction set will be shown in section 3. The final section describes a retargetable instruction simulator to evaluate the performance of our designed architecture.

2. Prototype Compiler of Arden

An experimental code generator generator is being developed and incorporated into GNU cc. This code generator uses simple specification language to facilitate general users constructing a compiler easily. *Arden* joins this retargetable code generator with GNU RTL generator as the prototype compiler system. Two major parts are involved:

1) GNU RTL Generator:

The work of this phase is translating the source language into an intermediate representation language (**RTL**), a parenthesized prefix expression form. Global optimizations are also achieved in this phase.

2) Retargetable Code Generator:

The current code generator in *Arden* accomplishes instruction selection by searching through instruction descriptions based on dynamic programming algorithm to get the best instructions with minimum cost. In the meantime, peephole optimization is done in part if combinational patterns are specified in the instruction descriptions. Part of peephole optimization will be delayed to post processor while tree patterns in instruction descriptions can not specify the characteristics of the targeted architecture.

In summary, the prototype compiler system of *Arden* modifies the machine dependent component in GNU cc and includes a post processor which is responsible for pipeline organization, special peephole optimization and instruction scheduling. It is a preliminary stage for *Arden* system to construct such a prototype compiler and to gather useful experience.

3. Instruction Description Language

Generally speaking, the processor description contains instruction descriptions and layout characteristics such as register usage, stack and memory management. Here we lay emphasis on instruction descriptions.

The chief goal of instruction description language in *Arden* is to make it easy to specify instruction set. The specification of instruction descriptions is a set of pattern-action rules. The syntax of an instruction rule in *Arden* is

```
%define_insn
@ Macro expression @
{Template }
@ Condition expression @
@ Cost @
{Action } %
```

where the entry between two @ is optional.

1) *Macro expression* defines the macro strings which will be expanded in other entries like template, condition or action. *Macro expression* facilitates users to define a subset of instructions which have similar *template, cost* and *action* specification in one define_insn rule.

2)*Template* is a parenthesized prefix expression representing a tree pattern.

3) *Condition expression* is a set of constraints applied to operands in *template*. The pattern matcher will make condition check after *template* is matched.

4) *Cost* is an optional simple assignment statement. When omitted, the pattern matcher in *Arden* assumes the default value. The default value is one, but can be redefined in another file named "machine.h".

5) *Action* is a C source code which returns the output object code for this rule. The output code is included by pattern matcher once this rule has been selected as part of the minimal cost instruction subset for the subject tree.

Macro expression has the following form:

```
VAR term = { string [,string  [ ...]  ]  }
        [ &term = { string [,string  [ ...] ]  } ]
        [ * term = { string [,string [ ...] ]  } ]
```

where

1) Term, a constant variable, represents a vector of string which is a set of cases applied in template, condition and action .
2) "&", a one-to-one connection operator, synchronizes two vectors of string while they are applied in template, condition and action.
3) "*", means a projection relation which expands two vectors of string to one combinational vector by row major.

Example 1

We can define four instructions ADD, ADDI, SUB and SUBI instructions of DLX instruction subset in one rule by specifying macro expression as follows:

```
 %define_insn
@ VAR operator  =  {"plus","minus"}
    *  mode      =  {"r","I"}
   &opcode  =  {"add" ,"addi", "sub", "subi"}
@
{ ( SI 0 r )   =  operator : SI ( SI 1 r )
                              ( SI 2 mode ) }
{ opcode  %0,%1,%2 }
%                                          ■
```

With the macro expression, the number of rules in instruction descriptions will be smaller.

The specification language has been used in describing DLX architecture. Experimental results show that it uses only 31 rules to describe DLX instruction set which was described originally in GNU by 144 rules, not including peephole optimization rules. There are significant reductions in the number of rules and the complexity to describe one architecture.

4. Improved Bottom-Up Matching Algorithm

The kernel of code generator generator is pattern matching routine. *Arden* adopts a modified bottom-up matching algorithm using numerical computation in the pattern matching of code generator generator. Before going through the pattern matching algorithm, we list the definitions of some terms which are used in complexity analysis and defined in [8].

patno	: the number of different patterns involved
patsize	: the size of the pattern forest
subsize	: the size of the subject tree
sym	: the number of symbols in alphabet Σ
rank	: the highest rank (arity) of any symbol in alphabet Σ
match	: the number of matches which are found

For the remainder of this section, complexity will be expressed in terms of the above defined terms. We assume the arity of all symbols is q-ary.

Bottom-Up Matching Algorithm The basic idea of bottom-up matching algorithm is to find, at each node in the subject tree, all patterns and all parts of patterns which match at this node[8]. The overhead of bottom-up matching algorithm comes from that the size of match set could be as large as $O(\, 2^{patsize}\,)$ and table size is $O(set^{rank} * sym\,)$. For calculating the match sets in step 1 and tables in step 2, the preprocessing time is $O(set^{(rank+1)} * sym * patsize)$. The complexity can be reduced if the pattern forest is simple. However, not every pattern forest matching problem can fit such constraints.

Top-Down Matching Algorithm Top-down matching algorithm regards each path from root to leaf in pattern tree as a string. The main loads of top-down matching method result from

1) constructing the pattern matching automaton, and

2) maintaining bit strings operation or counter at each node. The preprocessing space is $O(patsize^2)$ and the matching time is $O(subsize + match)$ [8]. But, to find all possible match sets in a subject tree, the top-down algorithm must involve a for-loop at each node to check all possible matches, starting at the node, for each pattern. Therefore, the physical complexity will be $O(subsize * patno + match)$.

Bottom-Up Matching Algorithm with Number Operations The idea for this algorithm originated from the observation that pattern matching problems will be simplified as long as a pattern tree could be reduced to a number. Then pattern matching will become a number comparison problem if each node of the subject tree is also represented by a number and traversed associated with number computation. We give each terminal and operator in intermediate language an identity number. Then the pattern trees will be translated into trees with identity numbers and calculated by an identical function from bottom to top. Finally, each pattern tree is labeled with its own computed value at root node. The pattern trees possibly match with the subtree in subject tree if the computed value rooted at this subtree is same as the computed values of pattern trees. The pattern matcher confirms the match by brute-force searching between the subtree and possible matched pattern trees. Our experiment with a random-generating pattern forest shows that different pattern trees with the same computed values are few if the *patno* is not larger than 10^3.

Example 2

Consider the following three binary pattern trees,

```
1. reg  <-   +           2. reg <-      +
            / \                        / \
         reg   reg                  mem   reg
                                    /
                                   reg
       cost = 1                      cost = 3

3. reg  <-   mem
             /
            +              cost = 2
           / \
        reg   reg
```

and the subject tree

```
      reg  <-     +
                 /    \
              mem      reg
               /
              +
            /    \
         reg      reg
```

Then we define the mapping function F as

F (op, left, right) = (op * left + right) mod prime

where the op, left, and right mean operator, left subtree, and right subtree, respectively. The prime is a prime number. To be more precise, we give the operators and terminals identity numbers as

reg : 2, mem : 4, + : 5

and prime is assigned to 41 arbitrarily. The null subtree is given as 1. It is straightforward to translate pattern trees to numerical trees. The first step is to calculate the value of each pattern tree using F from bottom to top. We obtain the results for three pattern trees as follows:

1)18 2) 23 3) 13

after applying the function F. Now, traverse the subject tree in postorder and apply the function F at the same time. ■

Definition 4.1:

Two pattern trees are *overlapping* if one of them is a subtree of another and a pattern forest is *overlapping* if there are any two patterns in it that are overlapping.

To pattern matching algorithm in code generator generator, the crisis is not finding subtrees which can be matched but getting all possible overlapping match sets. The reader will find that the overlapping factor for pattern forest is critical to matching time complexity in later discussion.

The primitive task of preprocessing routine is to traverse each pattern tree and record the height of it. The next task is to traverse again and to apply function calculation at all pattern trees in height sequence, from the pattern tree with shorter height to longer one. In the meantime, the routine must check the overlapping at each node of current pattern tree by looking at the previous computed pattern trees. When overlapping occurs, the number at this node will be replaced by the identity number of the replacement terminal which associated the overlapped pattern tree. Overlapping relation is also recorded at once. In example 2, pattern #3 contains an overlapping subtree at node "+" and the number will be replaced by 3. This means the computed value of "reg" which is the replacement node of pattern #1. It is the reason why the result number of pattern #3 is 13 (4 * 3 + 1). We illustrate the details of preprocessing phase in the following routines.

```
pre-process (Forest) {
    for each pattern tree $t_i$ in Forest
        height [i] <-  visit_height ( $t_i$ )
    sort Forest in height sequence
    for each pattern tree $t_i$ in Forest
        result [i] <-  visit_number ( $t_i$ )
}
int visit_height (tree) {
    if  tree is null
        return 0
    return (1 + max (visit_height (tree.left),
                     visit_height(tree.right)))
}
int visit_number (tree ) {
    if  tree is null
        return  null.id
    result = F (op, visit_number (tree.left),
                    visit_number (tree.right) )
    for all computed pattern $t_j$
        if result=result[j]  and true matching{
            record the match at $t_j$
            return $t_j$.replace.id
        }
    return result
}
```

The context of traversing subject tree routine is the same as visit_number() except for the part of match handling. For finding all possible matching covers for the subject tree, it must consider both replacement and non-replacement at matched node. When replacement is made, the match subtree will be rewritten into the left-hand side of matched pattern tree, for instance, the rewriting tree of pattern #1 in example 2 is "reg". If non-replacement is considered, record the match and go on the traversal. But, replacement is necessary when the matched pattern does not cover other pattern. As Twig[1] did, if multiple tree patterns match at one node, cost will determine which one is selected. The possible match cover trees of example 2 are shown in Figure 2. Between the two cover trees, the second cover tree treats the match at "mem" node as non-replacement and match at root in final. It is clear that first match set will be selected.

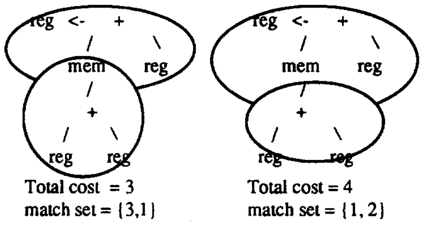

Fig. 2 Two possible cover trees for Example 2

In the worst case scenario, the preprocessing time is $O(patsize * patno)$ and matching time is $O(subsize * patno * 2^{match})$. However, the complexity will reduce to $O(subsize * patno)$ if there is no overlapping in pattern trees. In RISC based architecture, it is scare that one instruction can cover another one. It means the matching time for RISC base architecture set is $O(subsize * patno)$. The benefits of this algorithm result from

1) little space required,
2) no constraint on pattern forest,
3) simple computation and easy to implement ,
4) guarantee of minimum cost cover is selected

without string automaton. Table 1 lists the comparison of three matching algorithms.

5. Retargetable Instruction Simulator

In *Arden*, an instruction level machine simulator is developed to evaluate the performance of architecture and to gather run-time information for analysis when designing a new architecture or compiler technology. A designer can simulate the system before it is built and use the results to improve the design.

Time is the measure of computer performance. User CPU time (not including the time waiting for I/O or running system call) for a program can be expressed as

CPU time = (CPU clock cycles) * (clock cycle time)

where

CPU clock cycles = CPI (clock cycles per instruction) * (Instruction count)

Because clock cycle time is dependent on hardware technology and circuit design and it can be simulated by other CAD tools, we use only CPU clock cycles to represent the computer performance in *Arden*. In addition

Table 1. Space and Time complexity for pattern matching algorithms

Method	Constraints	Preprocess_time	Match_time	Preprocess_space	Match_space
Bottom_up with naive_preprocess	none	$O(set^{rank+1} * sym * patsize)$	$O(subsize + match)$	$O(set^{rank+1} * sym)$	$O(set^{rank} * sym + subsize)$
Bit_operation Top_down	none	$O(patsize)$	$O(subsize * patno + match)$	$O(patsize^2)$	$O(subsize * patsize + patsize^2)$
Modified bottom_up	non-overlapping pattern forest	$O(patsize * patno)$	$O(subsize * patno)$	$O(patno)$	$O(subsize)$

to CPI and instruction count, the simulator can gather some execution-time results like instruction level parallelism, time and frequency distribution of instructions, memory reference location, register usage, and branch results to improve the design. The target machines of our simulator are limited on RISC machines and include multi-issue processors called superscalar.

When developing a retargetable simulator, we need to define a method to describe a new architecture. There are two major requirements, the first is that it must be as simple as possible when describing a general design. If description is harder than writing a new simulator, the retargetable simulator will fail. The second is that the description can't limit the creation of new architecture. If new designs can not be described in advance, the retargetable simulator will be useless. But these two requirements sometimes conflict. We decided to employ object oriented technology to solve this problem. Object oriented programming can transmute the software design into choosing bricks, just as hardware design is now a matter of selecting integrated circuits. This new concept has advantages to modify and to reuse. Our work happens to be simulating the behavior of a hardware system and we can use this concept straightforwardly. A general design is described by combining the basic components, and a special design can be described by adding some special components or modifying the old ones.

The simulator is divided into four parts, namely, target machine, result handler, command interpreter and user interface. The target machine is the kernel of our simulator. It simulates the execution of a computer. The development of the target machine simulator includes three jobs, defining the structure of the target machine and implementation of basic functions, defining a description language, and writing a precompiler to translate the description language to programming language.

The components of the target machine are:

1)System call handler: system calls like *getc* or *open* are hard to be performed in an instruction level simulator. They are handled here and the results will be returned to the target machine.

2)Memory module: includes memory management to maintain the structure of computer memory and performs basic functions including *memory read, memory write, load file, dump memory.*

3)Registers module: maintains the structure of register file and performs basic functions including read register, set register

4)ALU module: simulates the behavior of arithmetic and logic unit and performs basic functions like *addition, shift, and.*

5)IF/PC module: maintains the program counter, simulates instruction fetch and handles the *branch* or *procedure call* instructions.

6)Decoder module: starts sequential actions according the instruction format description and instruction behavior description.

A machine description was defined by three parts. Machine configuration description defines some processor parameters like the number of registers, memory ports, functions units, instructions fetched per cycle and data width. Instruction format description defines the tokens of instructions and the fields of each instruction. It performs lexical analysis to scan an instruction and to decode it. Instruction behavior description describes the behavior and the operation latency (the delay cycles before the result is valid) of an instruction. The architecture designer can simulate a processor and does not have to worry about the hardware design in this way.

6. Conclusion

This paper presents an architecture developing system environment and a modified algorithm to find the optimal match pattern set for one subject tree with reduced space complexity considerably.

Reference

1. Aho, A. V., and M. Ganapathi, and S. W. K. Tjiang, "Code Generation Using Tree Matching and Dynamic Programming," *ACM Trans. Program Lang. Syst.*, Oct.1989, pp. 517-561.
2. Aigran, P., Graham, S., Herry, R., Mckusick, M., and Pelegri-Llopart.E, "Experience with a Graham-Glanville Stype Code Generator," *Proceeding of the ACM SIGPLAN Symposium on Compiler Construction, ACM SIGPLAN Notices 19, 6,* June 1984, pp. 13-24.
3. Cattell, R. G. G., "Automatic derivation of code generators from machine descriptions," *ACM Trans. Program Lang. Syst, vol. 2, No. 2,* Apr. 1980, pp. 173-190.
4. Emmelmann H. S., F. W. Landwehr, "BEG- A generator for Efficient Back Ends," *Proceeding of ACM Conference on Language Design and Implementation,* June 1989, pp. 227-237.
5. Fraser, C. W, "A Language for Writing Code Generators," *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation.* June, 1989., pp. 238-245.
6. Giegerich, R, "On the Structure of Verifiable Code Generator Specifications," *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation.* June, 1990, pp. 1-8.
7. Giron-Sierra, J. M., and Gomez-Pulido, J. A., "Doing Object-Oriented Simulations: Advantage, New Development Tools," *The 24th Annual Simulation Symposium,* 1991.
8. Hoffman, C. W., and O'Donnell, M. J, "Pattern Matching in Trees," *J. ACM 29, 1,* 1982, 68-95.
9. Hostetler, L. B., and Mirtch, B., "DLXsim- A Simulator for DLX," *Tech. Report,* Oct., 1990.
10. Lipsett, R., Schaefer, C., and Ussery, C.,"VHDL: Hardware Descritpion and Design," *Kluwer Academic Publishers,* 1989.
11. Stallman, R. M., "Using and Porting GNU CC," *Free Software Foundation, Inc,* 1991.