

ARDEN --- ARCHITECTURE DEVELOPMENT ENVIRONMENT

Feipei Lai, Shu-Lin Hwang, Tzer-Shyong Chen, Chia-Rung Hsieh
Dept. of Computer Science and Information Engineering
& Dept. of Electrical Engineering National Taiwan University
Taipei, Taiwan, China

ABSTRACT

This paper describes *Arden*, which is being developed to help architecture design. *Arden* includes a retargetable compiler and a back-end simulation tool that uses the concepts of object-oriented programming (OOP) to achieve model reusability. The code generator in the *Arden* compiler uses a tree pattern matching method for instruction selection. An experimental bottom-up matching algorithm that reduces the pattern matching to a numerical computation problem can reduce the space complexity and the search time. A useful instruction description language has been proposed to simplify the architecture specifications. We have implemented the DLX architecture with only 49 rules. *Arden* has been able to output DLX assembly code and has the same performance as GNU cc output.

INTRODUCTION

A quantitative approach in computer architecture [9] has been successful at providing critical information in architecture designs. The task of designing a new architecture would have many aspects, including instruction set design, functional organization, logic design, and implementation. Once a set of functional requirements has been established, the architect must try to optimize the design. A flexible compiler and reusable simulation tools will usually be useful to aid the architect in his task. There have been some retargetable compilers [1, 2, 3, 5] which attempted to improve the speed of generated code-generator and the quality of output code in order to verify the set of functional requirements for desirable machines. But, for a retargetable compiler, another critical component is the specification facilitation of machine-dependent parts. The major goals of *Arden* are to reduce the complexity of describing architecture characteristics, to eliminate the gap between the compiler and the architect, and to integrate the compiler with the simulator into a flexible environment.

Some of the developed code generator generators have the defect of targeting to one specific machine[6]. *Arden* adopts the instruction oriented description. Once the architect decides the instruction set, he can design straightforwardly the instruction description. The code generator in *Arden* uses dynamic programming technology to get optimal code as Twig[1] does and

feedbacks the possible cover tree for every node of an unmatched pattern tree to help architect modify the instruction description. Section 4 illustrates the pattern matching algorithm of *Arden*, and some comparisons between *Arden* and the alternatives.

An instruction level simulator can evaluate the performance of an architecture and to gather run-time information for analysis when designing a new architecture and compiler. Since the architect may refine the architecture frequently, a simulator should also support these refinements. We use the concepts of object-oriented programming (OOP) to achieve simulator reusability[4]. This will reduce the time of the development cycle.

Fig. 1 presents an overview of *Arden*. There are four critical components shadowed in Fig. 1:

Code generator generator: A code generator generator parses the RTL representation and makes instruction selection by searching through the instruction description of architecture description. Then it can output the object code or assembly code for the target machine.

Architecture descriptions: There is two parts to the architecture description:

(1)instruction description: The instruction description is specified using instruction description language[6].

(2)layout description: A set of constant definition contains memory, register and stack layout.

Processor object model: The reusable model makes simulator easy to redesign.

Instruction simulator simulates the behavior of the object code and gives performance information.

In section 2, we will review the instruction description language used in *Arden* and a summary of the prototype compiler system in *Arden* is given in section 3. Section 4 illustrates the pattern matching algorithm of *Arden*. Section 5 describes a reusable instruction simulator to evaluate the performance of our designed architecture. The final section will give some results of the DLX design.

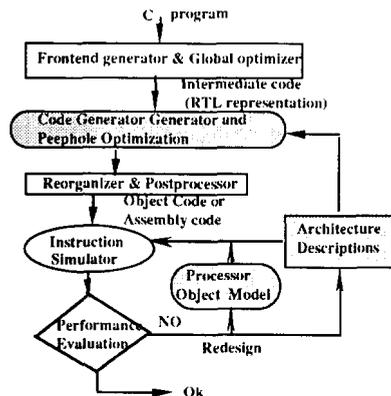


Fig. 1 Architecture Development Environment

INSTRUCTION DESCRIPTION LANGUAGE

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 }
{ Instruction format }
{ Operation macro } %

```

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 defining 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 of RTL representation.
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 value is set at one by default, but it can be redefined in another file named "machine.h".
5. *Action* is a C source code which returns the output object code or assembly code for this rule. The output code is written to output file by code generator once this rule has been selected as part of the minimal cost instruction subset for the subject tree.
6. *Instruction format* is the information concerning opcode, instruction type, and latencies about the Action.
7. *Operation macro* is a set of the executing functions of the Action. Our instruction simulator depends on the macro to trigger the operations of the Processor Object Model.

Example 1

Four instructions ADD, ADDI, SUB and SUBI instructions of DLX instruction subset can be defined in one rule by specifying macro expression as follows:

Rule 23:

```

%define_insn
@ VAR macro_type = {"r", "I"}
* macro_operator3 = {"plus", "minus"}
& macro_opcode = {"add", "addi", "sub", "subi"}
@ (macro_operator3 : SI (r SI 1)
(macro_type SI 2) )

@ cost = 1 @
{ "macro_opcode %r0,%r1,%m2 " }
{ opcode : 0x20 0x20000000 0x22 0x28000000
type : R I R I
kind : ARITH
latency : 1 1 1 1 }
{ IPU.arith(r,r,+) IPU.arith(r,i,+)
IPU.arith(r,r,-) IPU.arith(r,i,-) }
%

```

IF the template of RTL is:

```

(set (reg:SI 3) (plus:SI (reg:SI 4)
(reg:SI 3)))

```

THEN macro_type: r and macro_operator3: "plus" are matched, and Action output "add r3, r4, r3."

PROTOTYPE COMPILER OF ARDEN

An experimental code generator generator is being developed and incorporated into GNU cc [12]. This code generator uses simple specification language (see section 2) to facilitate general users constructing a compiler easily. Arden combines this retargetable code generator with GNU RTL generator as a prototype compiler system. Two major parts are involved:

1. GNU RTL Generator:

Rather than using a fixed intermediate language and compiling it for a variety of machines, GNU cc instead defines a representation called RTL, for register transfer language. The idea of using RTL and some of the optimizations came from the Portable Optimizer (PO), written at the University of Arizona [11]. RTL is usually written in a LISP-like notation; a typical addi insn looks like:

```

(insn 32 30 33 (set (reg :SI 14)
(plus:SI (reg:SI 14)
(const_int 8))) 84 (nil)
(nil))

```

The work of this phase is translating the source language into a RTL file. Global optimizations and register allocation 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.

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.

IMPROVED PATTERN MATCHING ALGORITHM

The kernel of the code generator generator is a pattern matching routine. *Arden* adopts a modified bottom-up matching algorithm using numerical computation in the pattern matching of code generator generator. The algorithm includes two phases - preprocessing and matching phase.

1. Preprocessing phase

The primitive task of the preprocessing routine is to traverse each pattern tree and to record its height. The next task is to traverse again and to apply the mapping function calculation on all the pattern trees in the height sequence, from the pattern tree with the shortest height to the longest one. In the meantime, the routine must check the overlapping at each node of a current pattern tree by looking at the previous computed pattern trees. We illustrate the details of the preprocessing phase in Algorithm 1.

```

pre_process(Forest)
{
  for each pattern tree ti in Forest do
    height[i]=visit_height(ti)
  sort Forest in height sequence
  for each pattern tree ti in Forest do
    result[i]=visit_number(ti)
}
int visit_height(tree)
{
  if tree is null then return 0
  else return(1+max(visit_height(tree.left),
    visit_height(tree.right)))
}
int visit_number(tree)
{
  if tree is null then return null.id
  result=F(op,visit_height(tree.left),
    visit_height(tree.right))
  for all computed pattern tj do
    if result==result[j] and true matching record
      the match at tj
  return result
}

```

Algorithm 1: Preprocessing Phase

2. Matching phase

The context of traversing subject tree routine is the same as visit_number() except for the match handling. To find all the possible matching covers for the subject tree, it must consider both replacement and non-replacement at the matched node. When replacement is made, the match subtree will be rewritten into the left-hand tree of the matched pattern tree. If non-replacement is considered, record the match and go on the traversal. As Twig did, if multiple tree patterns match at one node, the cost determines which one will be selected. The details of the matching phase algorithm are given in Algorithm 2.

```

int pattern_matcher(RTL code)
{
  traverse the subject tree in postorder and
  apply the number computation
  if match at current node record the match
  node information in the match parsing stack
  postprocess()
}

int postprocess()
{
  for each element in the match parsing stack do
  {
    do replacement and go on the computation
    from the replacement node till to the root.
    if match at the root then save the match set
  }
  if there are multiple match sets then choose the
  one
  with the minimum cost
}

```

Algorithm 2: Matching phase

The benefits of this algorithm result from

1. little space required,
2. no constraint on pattern forest,
3. simple computation and easy implementation,
4. guarantee of the minimum cost cover being without a string automaton.

The specification language has been used in describing DLX architecture. Experimental results show that it uses only 49 rules to describe DLX instruction set which was described originally in GNU by 125 rules. In addition, the size of the description file in *Arden* is much smaller than it is in GNU cc originally. The comparison between *Arden* and GNU cc is given in Fig. 2. There are significant reductions in the number of rules and the complexity to describe one architecture.

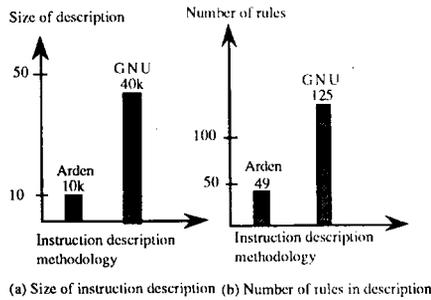


Fig. 2 Instruction description comparison between Arden and GNU
(a) Size of instruction description (b) Number of rules.

Our experience with the implementation of a pattern matcher suggests that the overhead of finding a minimum cost instruction selection is mainly in the replacement of match nodes. Table 1 shows that the number of replacement nodes is less than 2. This means that the overhead of the backing replacement node is reasonable. In the preprocessing time for DLX instruction description, the top-down method uses 1900 ms and the number-reduced algorithm uses only 83 ms. Table 2 gives the comparison between the top-down method and the number-reduced method in the matching time.

Table 1. Number of Match replacement nodes (The benchmark programs are a part of GNU cc.)

Test program	tree.c	topev.c	jump.c	loop.c
Number of replace nodes	2776	4996	3783	15642
Number of RTL instruction	1798	3113	2598	10009
Average replaced nodes	1.54	1.60	1.456	1.56

Table 2. The comparison between Top-down method and Number-reduced method in matching time for four programs.

Test program	tree.c	topev.c	jump.c	loop.c
Number-reduced (milli-sec)	5666	2717	9550	6016
Top-down(milli-sec)	20649	10010	33542	24949
Time ratio	.274	.269	.285	.241

REUSABLE INSTRUCTION SIMULATOR

A designer can simulate the system before it is built up and use the results to improve the design. Time is the metric of computer performance. The CPU time (excluding the time waiting for I/O or running system call) for a program can be expressed as followed.

$$\text{CPU time} = (\text{CPU clock cycles}) * (\text{clock cycle time})$$

where

$$\text{CPU clock cycles} = \text{CPI (clock cycles per instruction)} * (\text{Instruction count})$$

The clock cycle time is dependent on hardware technology and circuit design. Besides it can be simulated by other CAD tools. Therefore, we use only CPU clock cycles to represent the computer performance in *Arden*. Our instruction simulator gathers 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 restricted on RISC machines and the multi-issue superscalar processors.

When developing a reusable instruction simulator, we need to define a method to describe a new architecture. We decided to employ object oriented technology to solve this problem. Object oriented programming (OOP) can transmute the software design into choosing bricks, just as hardware design is now a matter of selecting integrated circuits. The object oriented approach is suitable for simulation, with important benefits for the software and project management: software re-use, modular and structured, divide-and-conquer strategy[8]. The principal idea associated with object oriented programming is that all items in the system are treated as "object." An object is either a "class" or an instance of a class. A class is that software module which provides a complete definition of the capabilities of members of the class. These capabilities are either provided by the procedures and data storage contained within the immediate class definition or inherited from other class definitions to which this class is related.

The simulator is divided into three parts, namely, target machine, result handler, command interpreter. The target machine is the kernel of our simulator. Our work happens to be simulating the behavior of the target machine which consists of basic components and special units. The development of the target machine simulator includes three jobs, defining the structure of the target machine, implementation of basic functions, and defining instruction format and operations associated with every instruction. We have constructed the instruction format and operations macro by instruction description (see section 2) and use OOP to construct the Processor Object Model for the new target machine. The object model of basic components can be re-used in next design cycle. The only thing the designer needs to do is to modify the object model of special components in the redesign cycle.

Fig. 3 shows the Processor Object Model and the basic components of it are:

1. System call class: system calls like *getc* or *open* are hard to be performed in an instruction level simulator.

They can be handled here and the results will be returned to the target machine.

2. Memory class: includes memory management to maintain the structure of computer memory and performs basic functions including *memory read*, *memory write*.
3. Registers class: maintains the structure of register file and performs basic functions including *read register*, *set register*.
4. ALU class: simulates the behavior of arithmetic and logic unit and performs basic functions like *addition*, *shift*. In RISC machine, ALU is divided into two parts, one is IPU and the other one is FPU. Two subclasses IPU and FPU can be defined and they inherit from ALU class.
5. IF/PC class: maintains the program counter, simulates instruction *fetch*.
6. Control/Branch class: handles the *branch* or *procedure call* instructions.
7. Decoder class: starts sequential actions according to the instruction format field and operation macro field of instruction description.

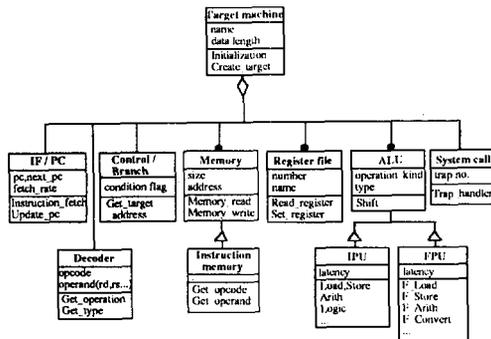


Fig. 3 Processor Object Model for simulating target machine

EXPERIMENTATION OF THE DLX

We have performed some experiments for DLX architecture using *Arden*. The instruction description has been implemented by using instruction description language (see section 2) and uses only 49 rules to describe DLX architecture. Eleven of these 49 rules are used for the arithmetic and logic instruction. Fourteen of them are used for the data transformation instruction. Fourteen of them are used for the floating point instruction and four of them are used for the special instruction (e.g. nop).

We input nine test programs into *Arden* and output DLX assembly code. Then the assembly code has been verified by *DLXsim* [10], and get the performance data (e.g. executing count, branch results). The result is quite similar to GNU cc's. The set of test programs includes *qsort.c* (quick sort problem), *8q.c* (eight queens problem), *shortpath.c* (salesman traveling problem), *HT.c* (Hanoi Tower problem), *kmp.c* (a simple test for the Knuth-Morris-Pratt Pattern Matching Algorithm), *sieve.c* (Eratosthenes Sieve Prime Number), *ts.c* (a printing out test), *ack.c* (a recursive computation), *wc.c* (a character processing program.)

CONCLUSIONS AND FUTURE WORK

This paper presents the development of a framework for *Arden*. It has used a modified matching algorithm to find the optimal match pattern set for one subject tree with reduced space complexity and has proposed an instruction description language to ease making the new architecture specifications. The simulator also depends on it to execute proper operations. We are developing the simulator by using OOP technology and are able to achieve simulator reusability.

There are many possible areas for future work. Next step, we will develop SPARC environment by *Arden*. In the future, we hope to add the post-scheduling algorithm into the system for superscalar processor design.

Reference

1. Aho, A. V., Ganapathi, M., and Tjiang, S. W. K., "Code Generation Using Tree Matching and Dynamic Programming," *ACM Trans. Program Lang. Syst.*, Oct. 1989, pp. 491-516.
2. Aigran, P., Graham, S., Herry, R., Mckusick, M., and Pelegri-Llopert, E., "Experience with a Graham-Glanville type 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. Pratt, B. D., Farrington, A. P., Basnet, B. C., Bhuskute C. H., Kamath, M., and Mize, H. J., "A Framework for Highly Reusable Simulation Modeling: Separating Physical, Information, and Control Elements," *The 24th Annual Simulation Symposium*, April, 1991, pp. 254-261.
5. Emmelmann, H. S. and Landwehr, F. W., "BEG- A generator for Efficient Back Ends," *Proceeding of ACM Conference on Language Design and Implementation*, June 1989, pp. 227-237.
6. Lai, Feipel, Tsaur, F., and Shang, R., "ARDEN --- ARchitecture Development ENvironment," *IEEE TENCON 92*, Nov., 1992, pp. 181-185.
7. 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.
8. Joes M. Giron-Sierra and Juan A. Gomez-Pulido, "Doing Object-Oriented Simulations: Advantage, New Development Tools," *The 24th Annual Simulation Symposium*, April, 1991, pp. 177-184.
9. Hennessy, J. L., and Patterson, D. A., "Computer Architecture a Quantitative Approach," Morgan Kaufmann Publishers INC, 1990.
10. Hostetler, L. B., and Mirtch, B., "DLXsim - A Simulator for DLX," *Tech. Report*, Oct., 1990.
11. Davison, W. J., and Fraser, C. W., "Register Allocation and Exhaustive Peephole Optimization," *Software - Practice and Experience 14*, Sep, 1984, pp. 857-865.
12. Stallman, R. M., "Using and Porting GNU CC," *Free Software Foundation, Inc.*, 1991.