ARES - Architecture REinforcing Superscalar

Yuh-Haur Lin, Feipei Lai, and Meng-Chou Chang
 Dept. of Computer Science and Information Engineering
 &
 Dept. of Electrical Engineering
 National Taiwan University
 Taipei, Taiwan, R.O.C
 Tel : 886-2-3635251, 886-2-3625252 Ext. 256
 Fax : 886-2-3638247
 e-mail : flai@leibniz.ee.ntu.edu.tw

Abstract

In the ARES, there are four major features. First of all, the ARES utilizes both static and dynamic scheduling methods. The instruction identifier bits (IDBs) are attached to each instruction [Smith 1990], except jump and branch, to indicate which basic block (BB) the instruction originally belongs to. Thus, the compiler can move instructions across the boundaries of BBs to get more instruction level parallelism. Secondly, we separate memory address calculation from memory access for Store instruction. In this way, we can divide the original two memory cycles of Store instruction for write-back cache into two separate cycles: MEM_C (memory check) in memory address calculation instruction (*sadd*) and MEM_W (memory write) in memory access instruction (*st*), therefore, there is no delayed slot between Store and Load instructions. And with Store Buffer, the memory address calculation instructions can be boosted up to the preceding BBs, then the memory access can be triggered by the real store instructions in the current BB. All these actions are scheduled by the compiler of the ARES with the other instructions. Thirdly, we divide the instruction of compare-and-branch into two steps: Compare and Test-then-branch. With this scheme and branch registers, we can combine the current BB with the following BBs (taken/untaken or both) into one BB to increase the schedulable instructions. Finally, we follow the same way as the IFU of the MARS system ([MARS 1989], [Maa 1989], [Tsai 1990]) to peep and absorb jump, so there is no delayed slot for jump in the ARES. Our architecture can have a 1.62 speedup, compared with the MIPS-X, with a simple extra hardware support.

I. Introduction

A. VLIW vs. Superscalar

For a given task *J*, the time to complete it can be formulated as follow:

$$\text{time}(\text{task } J) = N * \text{CPI} * T$$

J: *N* : the number of instructions to complete the task

CPI : clock cycles per instruction.

T : clock cycle time.

A VLIW (Very Long Instruction Word) machine tries to

divide *N* by some factor *F*. The factor is decided by how many sub-operations combined in one long instruction word. On the other hand, the superscalar machine issues multiple instructions in a single clock cycle. That is, it divides *CPI* by a factor *S* which is the number of instructions issued in a single clock cycle.

The techniques of instruction scheduling adopted by VLIW and superscalar are somewhat different. The VLIW [Colwell 1987] uses static scheduling. It requires a very powerful compiler to achieve the compact instructions. The technique called Trace Scheduling [Fisher 1981] was proposed to do global compaction of instructions. The drawbacks of VLIW are code expansion, program compatibility, low code density, and a very complex compiler. But VLIW can schedule instructions by considering a wide range of codes, so it can get more instruction level parallelism and do a better arrangement of code sequences to utilize hardware more efficiently. Another advantage of VLIW is that its hardware would be simpler than that of superscalar.

Superscalar uses dynamic scheduling to schedule instructions, so it can get some run time information to schedule instructions (for example, dynamic renaming). But according to [Johnson 1989], we need four major hardware features: out-of-order execution, register renaming, branch prediction, and a four-instruction decoder to achieve only nearly twice the performance of a scalar processor. The complexity of the hardware obviously can not justify the performance. This is the most serious problem of superscalar architectures.

From the above discussion, we can find each of VLIW and superscalar has its own advantages and disadvantages. How to make good tradeoffs between VLIW and superscalar has become a very important issue. Another important issue is that we must carefully consider the cycle time. We should not sacrifice the cycle time to achieve more operations per cycle such that the enhancement can not justify the sacrifice.

B. Control and data dependencies

i) Control dependency

The penalty of control dependency has become more and more intolerable as the techniques go toward more deeply pipeline and more operations per clock cycle. There had been so many ways to solve the problem of control dependencies [Lee 1984, McFarling 1986]. One of the most popular methods is prediction. We can do prediction statically or dynamically. In [Smith 1990], the hardware maintains a duplication of register file called shadow register to hold the uncommitted results. So they can use static prediction to boost instructions from the

predicted BB. The advantages of boosting are: (1) it can attain the same hit ratio as the dynamic prediction using profiling, but it does not need a BTB (Branch Target Buffer), and (2) the boosted instructions from the predicted BB can be placed in proper position of the current BB by static scheduling. The only drawback of boosting is that we need some extra bits in instruction to distinguish between the boosted instructions and the normal instructions. And we need more bits to locate and access register file.

ii) Data dependency

Let $R(k)$ be the set of registers that the instruction k doing a read and $W(k)$ be the set of registers that the instruction k doing a write. Now consider two instructions i and j , with i occurring before j . There may exist three types of data dependencies between them [Patterson 1990].

(1). true dependency (RAW)

If $W(i) \cap R(j) \neq \emptyset$ then we say there is true dependency between instructions i and j .

(2). antidependency (WAR)

If $R(i) \cap W(j) \neq \emptyset$ then we say there is antidependency between instructions i and j .

(3). output dependency (WAW)

If $W(i) \cap W(j) \neq \emptyset$ then we say there is output dependency between instructions i and j .

We can use register renaming to avoid both the WAR and WAW data dependencies. In superscalar, the dynamic renaming technique is adopted.

C. The problem of Load/Store

The only way of RISC-typed processors [Patterson 1982] to communicate with the outside world (cache or memory) is via Load/Store instructions. There are two issues having something to do with Load/Store: (1) the Load instructions should be executed as early as possible, and (2) the Store instructions should not prevent the following Load instructions from execution if they can be executed. In pure dynamic scheduling, they can do Load/Store reorganization to achieve a better performance, but the cost is too high. We make a better trade-off in the ARES. With shadow register, we can boost the Load instructions upward into the preceding BB. And we separate memory address calculation from memory access for Store instruction. This arrangement has the following advantages: (1) the address calculation instructions can always be boosted and are more easily to be scheduled with other instructions, (2) with Store Buffer, we can boost the following Load instructions and the first part of the divided store ahead the second part of the divided store. In other word, we can resolve the issue (2) described above to some extent, and (3) there is no delayed slot between Store and Load instructions. Originally, if we adopt write-back cache, the Store instruction needs two memory cycles: one for reading and checking the cache line status, and the other for storing the data. Thus, if Load instruction follows Store instruction immediately, we should insert delayed slot between them to avoid confliction. That is, the Load instruction would be placed in a latter slot. We attack this problem by using *sadd* and *st* instructions. *Sadd* fetches the stored data from register file, then places it in the Store Buffer. In the same time, it calculates the store address and places the address in the SARF. It also reads the cache line status in the MEM_C cycle. While *st* instruction acts as a signal of the

compiler to inform the Store unit to write the specified data. The only restriction is that *st* can not be placed right immediately after *sadd*. So it is very easily for the compiler to schedule *st* with other instructions. And Load can be placed right after *sadd* and *ld*. When we scheduled program, we found the separation of memory address calculation and memory access for Load instruction will incur additional cycles. In order not to delay the Load instruction, we still treat it in traditional way.

D. Optimization Compiler

No matter what kind the processors are (RISC-type, superscalar, or VLIW), they cannot achieve very good performance without the optimizing compiler. There is one thing we should keep in mind: the less the hardware is exposed to optimization compiler, the more easily for us to keep programs compatible. The compiler of the ARES is only informed the limitation of scheduling in one slot (listed in TABLE 1) by the hardware, therefore, the ARES can still execute without compiler optimization (in single instruction mode). In section II, we introduce the hardware architecture, ISA, and the instruction pipeline of the ARES. Then we will show how the optimizing compiler works to enhance the performance in section III. Results and conclusion are given in the final section.

II. The Architecture of the ARES

A. Hardware of the ARES

The address and data bus of the ARES is shown in Figure 1. The hardware of the ARES is shown in Figure 2. It consists of an IFU & Branch Resolver, an on-chip instruction cache, a simple Store unit, two ALUs, and some special register files. The size and port number of these register files are described in Table 2. The untaken register file maintains the result of the boosted instructions from the BB which will be executed if branch is untaken, while the taken register file maintains the result of the boosted instructions from the BB which will be executed if branch is taken. The comparison results of branch instructions are held in Branch Register File. With Branch Register File, we can hold the comparison results of the branch instructions which belongs to the following BBs, therefore, we can handle these branch instructions simultaneously. The SARF hold the address for the Store instructions. In order to achieve precise interrupt, the Store instruction cannot be boosted across the boundary of BBs. But in the ARES, we divide the Store instruction into address calculation instruction (*saad Src1,X,SARF#,Src2*) and the real store instruction (*st SARF#*). The instruction *saad Src1, X, SARF#, Src2* does the following actions: "SARF# <- Src1 + X" and "Store Buffer[#] <- Src2". So the value to be stored will be placed in Store Buffer until the real store instruction saving it to memory. In this way, we have: (1) the instruction *sadd* can be boosted up, and (2) the Load instructions following the real store instruction can be placed in the position above the real store instruction, but below the instruction *sadd* that is corresponding to that real store instruction.

B. The ISA (Instruction Set Architecture) of the ARES

The current ISA of the ARES is adopted from MIPS-X processor [Chow 1989] with some modifications for the special

needs of the ARES. The differences incur in branch instructions and Store instructions. The branch instruction has only one operand. The operand is compared with 0 to decide whether the branch is taken. We can use some simple combinational circuits to do this job without ALU. For Store instructions, we do not include ALU operations to calculate operand address (it can be achieved by the *sadd* instructions before the Store instructions). Table 3 shows the corresponding instruction codes between the MIPS-X and the ARES. The instructions *clb*, *movtob*, *movfrb*, *jumpb*, *jumpr*, *call*, *ret* and *rfe* are unique for the ARES to handle jump peeping and absorbing (MIPS-X uses *jspci* to handle jump). Their functions are described as follows:

(1) <i>clb</i>	BaseReg <- 0
(2) <i>movtob Src</i>	BaseReg <- Src
(3) <i>movfrb Dest</i>	Dest <- BaseReg
(4) <i>jumpb N</i>	PC <- BaseReg + N
(5) <i>jumpr N</i>	PC <- PC + N
(6) <i>call N</i>	RAS <- PC + 3; PC <- BaseReg + N
(7) <i>ret</i>	PC <- RAS (return)
(8) <i>rfe</i>	PC <- RAS; restore machine state (return from exception)

BaseReg (Base Register) and RAS (Return Address Stack) are the special registers in the IFU and Branch Resolver [MARS 1989]. The other two unique instructions for the ARES are *ld ras* (load RAS) and *st ras* (Store RAS). When RAS is full or empty, the ARES uses these two instructions to store or load RAS.

The original instructions *jpc* and *jpcrs* (the MIPS-X uses these instructions to handle exception) are excluded from the ARES.

C. The Instruction Pipeline

The instruction pipeline of the ARES, consisting of 5 stages, is shown in Figure 4. For the first stage, the ARES will fetch four instructions from external instruction cache to fill the instruction buffer. The functional decoder will scan the first three instructions and decide what functional units they need. In the same time, the ARES will peep the fourth instruction to see if it is a jump instruction to be absorbed (the same way as the IFU of the MARS system). Figure 6 shows the instruction decode window & jump peeping. The timing of jump instruction peeping/absorption is shown in Figure 5.

After instructions passing through the first stage, we can dispatch them into proper functional units. The lower half part of Figure 4 shows the different types of instruction pipeline in different functional units. The timing of delayed slot branch is shown in Figure 7. In Figure 8, we show one-booster-slot Store. For the boosted slots, we can boost the address calculation instruction into the upper BB or delay the *st* instruction to fill the empty slot. Figure 9 shows that the Load instruction can be placed right after *sadd*.

III. Optimization Compiler

As the shadow part of Figure 10 describes, the compiler of the ARES consists of five functional modules: MAPPING, Partition BB, BOOSTED, Code Optimization, and Global Optimization. The latter two modules are optional. The compiler can still work without them. Code Optimization tries to eliminate the redundant code of the program, to move loop-invariant instructions out of the loop, and to do some other things just like the traditional compiler does. Global Optimization can move

code across boundaries of BBs without the help of hardware to decrease the instruction steps [Tokoro 1978]. The major parts of the compiler of the ARES are as follows: (1) MAPPING uses TABLE 3 as a reference to transfer the MIPS-X code to the ARES code, (2) Partition BB tries to divide the program into a set of BBs and establishes the corresponding control graph. For each BB, Partition BB will assign a unique number to identify it, and (3) BOOSTED will boost the needed instructions from the predicted BB and schedule these instructions with the instructions of current BB according to the limitation listed in Table 1. BOOSTED will deal with those BBs in the most likely executed path first. The algorithm of BOOSTED is listed as follows:

INPUT:

A set of BBs (N) in a procedure (the entry BB number is, 1) with control dependence graph established for them by Partition BB.

TARGET:

To generate a new set of BBs. Each of them has already been scheduled and built from the set of old BBs, but both of them are semantically equivalent.

FUNCTION AND FLAG:

done[i]: the flag to indicate whether the BB *i* has arranged the boosted or not (**done[0]** = TRUE).

push(i): push the BB *i* to the stack.

pop: get a BB from the top of stack.

stack_empty: a flag to indicate whether the stack is empty or not.

all_done: a flag to indicate whether all BBs have been traversed.

prediction(i): it will return the BB number to which the control flow will go from the BB *i*, if BB *i* has no BB following or it has transferred the control flow to another procedure, then return 0.

establish DAG(i,j): establish the DAG for the combination of BB *i* and BB *j*, then return the list.

scheduling(l): schedule the dag list *l* according to the data dependency and the limitation of the ARES, then return the result.

boosted(s,k): according to the result of **scheduling s** to decide how many instructions of BB *k* should be boosted up, then find all the preceding BBs of BB *k* and boost these instructions upwards.

following_BB(i,j): return all the BBs following the BB *i* except for BB *j*, if there is no BB following then return 0.

procedure_in_BB(INPUT): return all the BBs which are the place the calling procedures come back except for BB 1.

ALGORITHM:

```

procedure BOOSTED ( INPUT)
{
  for i = 1 to N do
    done[i] = FALSE;
  push(1);
  while (! stack_empty) or (! all_done) do
  {
    if (! stack_empty)
    {
      j = pop;
      if ((k = prediction(j)) != 0)
      {
        boosted(scheduling(establish_DAG(j,k)),k);
      }
    }
  }
}

```

```

done[j] = TRUE;
for all l in [following BB(j,k)] do
  if (done[l] == FALSE) push(l);
  if (done[k] == FALSE) push(k);
}
else
  for all l in [Procedure in BB(INPUT)] do
    if (done[l] == FALSE) push(l);
  }
}

```

IV. Results and Conclusion

We scheduled some programs following the algorithm of the ARES compiler by hand (the instruction-level simulator is still being developed), and compared the program steps between the MIPS-X (generated by the MIPS-X compiler) and the ARES. A program step is a set of instructions that can be executed simultaneously (e.g., 3-4 for the ARES, 1 for the MIPS-X). This static evaluation is unfair for the ARES, because (1) boosting will generate some redundant codes, such codes are useful in run time execution, and (2) the boosted operation with Store Buffer, combination of BBs, and the ability to peep and absorb jump can increase the software pipelining efficiency within a loop. So in dynamic evaluation we can expect a better speedup than the MIPS-X. Although in such condition, the ARES can achieve 1.62 speedup in average for four testing programs. The results are shown in Table 4. From the results, we have the following observations: (1) the *ackerman* and *fibonacci* are recursive programs. In the present algorithm of BOOSTED, the optimized unit is a procedure. So when the program calls itself recursively, BOOSTED will treat the call instruction as a break point. The result is that many isolated BBs will exist in the program and the ARES can do nothing to enhance the performance except for the hardware redundant and jump absorbing, (2) the prediction of the current BOOSTED is always taken, so BOOSTED sometimes make efforts on the wrong path. The example is bubble sort. In fact, if the main path of the bubble sort, to be scheduled first, changes, the performance would be better. In the insertion sort, the BOOSTED has the right prediction.

According to the above discussion, we can have the following improvements: (1) do control flow analysis in the program, check out all the loops and treat the recursive programs specially, then call the BOOSTED to schedule the program from inner loops to outer loops, (2) use profiling to enhance the *prediction hit* rate, so the BOOSTED can really schedule the most likely executed path with the following features of the ARES: boosted, Store Buffer, combination of BBs, and peeping and absorbing jump.

V. Reference

- [Chow 1989]
Paul Chow, "The MIPS-X RISC Microprocessor," KLUWER ACADEMIC PUBLISHERS., 1989.
- [Colwell 1987]
R.P. Colwell, R.P. Nix, J.J. O'Donnell, D.B. Papworth, and P.K. Rodman, "A VLIW Architecture for a Trace Scheduling Compiler," Proc. 2nd Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS II), pp. 180-192, Oct.1987.
- [Fisher 1981]
J. A. Fisher, "Trace scheduling : A technique for global microcode compaction," IEEE Trans. Comput., vol. c-30, pp.478-490, July 1981.
- [Johnson 1989]
William M. Johnson, "Super-Scalar Processor Design," Technical Report No. CSL-TR-89-383, June 1989.
- [Lee 1984]
J. K. F. Lee, A. J. Smith, "Branch Prediction Strategies and Branch Target Buffer Design," IEEE computer, Jan. 1984, pp. 6-22.
- [MARS 1989]
G-S Jang, et al., "MARS-Multiprocessor Architecture Reconciling Symbolic with Numerical Processing" Int'l Symp. VLSI Technology, Systems, and Applications., May 1989, pp.365-370.
- [Maa 1989]
Y.-C. Maa, "Designing Instruction Fetch Unit for the MARS System," Master Thesis, Computer Science Div., Dept. of Electrical Engineering, National Taiwan University, June 1989.
- [McFarling 1986]
S. McFarling, J. Hennessy, "Reducing the Cost of Branches," Proc. 13th Symposium on Computer Architecture, Jun. 1986, pp. 396-403.
- [Patterson 1982]
David A. Patterson and Carlo H. Sequin, "A VLSI RISC," Computer, v.15, no.9, pp. 8-20, Sept. 1982.
- [Patterson 1990]
David A. Patterson, John L. Hennessy, "Computer Architecture A Quantitative Approach," MORGAN KAUFMANN PUBLISHERS, INC., 1990.
- [Smith 1990]
Michael D. Smith, Monica S. Lam, and Mark A. Horowitz, "Boosting Beyond Static Scheduling in a Superscalar Processor," Proceedings of the 17th Annual International Symposium on Computer Architecture (1990), pp.344-354.
- [Tokoro 1978]
M. Tokoro, T. Takizuka, E. Tamura, and I. Yamaura, "Towards an efficient machine-independent language for microprogramming," in Proc. 11th Annu. Microprogramming Workshop, SIGMICRO, 1978, pp. 41-50.
- [Tsai 1990]
W.-C. Tsai, "The Design and Implementation of Instruction Fetch Unit (IFU) of the MARS System," Master Thesis, Dept. of Computer Science and Information Engineering, National Taiwan University, June 1990.

branch	branch or jump	branch or jump
branch	branch or jump	ALU
branch	branch or jump	Load or Store
branch	ALU	branch or jump
branch	Load or Store	branch or jump
branch	Load or Store	ALU
branch	ALU	ALU
Store	ALU	ALU
Store	Load	ALU
jump		

Note : (1) If two ALU instructions are scheduled in one slot, then only one of them can be the Store Address Calculation instruction. The test instruction also follows this restriction.
(2) If there exists two or three branch (or jump) instructions in one slot, then the second position filled with the branch instruction from the taken BB, the third position filled with the branch instruction from the untaken BB.

Table 1. The limitation for optimizing compiler to schedule instructions in one slot

REGISTER FILE NAME	SIZE (word)	PORT NUMBER
Branch Register File	4	one write port three read ports
untaken, current, or taken register file	32	two write ports five read ports
Store Address Register File	15 (5-5-5)*	one write port one read port
Store buffer	15 (5-5-5)*	one write port one read port

*(8-8-8) : (untaken-current-taken)

Table 2. The register files of the ARES

Figure 1. The address and data bus of the ARES (MARS II)

Figure 2. The block diagram of the ARES

Figure 3. The instruction format of the ARES

Figure 4. Pipeline of the ARES

Figure 5. Jump instruction peeping/absorption

Figure 6. Instruction decode window & peeping jump

Figure 7. One-Delayed-Slot Branch

Figure 8. One-Boosted-Slot Store

Figure 9. Load can be placed right after sadd instruction

Figure 10. The functional flowchart of the optimization compiler of the ARES

MIPS-X Code	ARES Code
branch Src1,Src2,Label	test Src1,Src2,RR# branch RR#,Label
st X[Src1],Src2	sadd Src1,X,SARF#,Src2 st SARF#

The jump and call does not show here.

Table 3. The mapping table for transferring the MIPS-X code to the ARES code

Evaluation Program	program / supercalar for ARES codes	static SPEEDUP (MIPS-X code / ARES code)		
		count all delayed slots	without reorganization	with reorganization
ackerman	1.76	1.78	1.68	1.55
fibonacci	1.78	1.71	1.57	1.43
insertion sort	2.54	2.54	2.18	1.96
bubble sort	1.63	1.85	1.69	1.54

Table 4. The static speedup (MIPS-X program steps / ARES program steps)