Microprocessing and Microprogramming 36 (1993) 205–213
North-Holland


# Exploiting multi-way branches to boost superscalar processor performance*

Yen-Jen Oyang

*Department of Computer Science and Information Engineering, National Taiwan University, Taipei, Taiwan*

Received 10 January 1993
Revised 7 April 1993
Accepted 13 May 1993

*Abstract*

Oyang, Y.-J., Exploiting multi-way branches to boost superscalar processor performance, Microprocessing and Microprogramming 36 (1993) 205–213.

This paper addresses exploiting multi-way branches to boost superscalar processor performance. The work presented in this paper comprises two conjunctive parts. The first part is a compiler technique called the SV (Shadow Variable) transformation. The second part is a new multi-way branch scheme developed in conjunction with the SV transformation. The SV transformation can transform program segments in which multi-way branches are originally not applicable into ones in which multi-way branches are applicable. The proposed multi-way branch scheme is able to carry out multi-way branches efficiently, especially for those derived from applying the SV transformation. An effectiveness evaluation shows the SV transformation and the proposed multi-way branch mechanism together can boost superscalar processor performance by 11–14%.

*Keywords.* Superscalar microprocessor; multi-way branch; instruction-level parallelism; instruction scheduling; control dependence.

## 1. Introduction

Superscalar microprocessors have emerged as the new-generation microprocessors beyond RISC [1]. However, for non-numerical applications, which most general-purpose microprocessors are developed for, the speedups that can be achieved with superscalar microprocessors are severely limited by the great number of branch instructions in the program, one out of every three to seven instructions [2,3]. Branch instructions cause two negative effects to superscalar processors. The first effect is the inducing of control dependence in the program, which, in turn, seriously limits the exploitation of instruction-level parallelism. The second negative effect is the penalty due to carrying out a branch operation. Though branches always cause some forms of penalty on pipelined processors, the effect is more severe to a superscalar processor. For a superscalar processor that executes $N$ instructions per clock cycle, the waste of one clock cycle due to a branch operation will result in a degradation of performance by the same percentage as the waste of $N$ clock cycles in a processor that executes one instruction per clock cycle.

In acknowledging the impact of branch instructions on superscalar processor performance, we investigated the idea of exploiting multi-way branches to reduce the number of branch operations in the program. This effort leads to the development of a compiler technique called the SV (Shadow Variable) transformation and a new hardware multi-way branch scheme. The SV transformation is proposed to promote the applicability of multi-way branches by transforming program segments in which multi-way branches are originally not applicable into ones

*This research was sponsored by National Science Council of R.O.C. under contract NSC 80-0408-E-002-15.

in which multi-way branches are applicable. The new multi-way branch scheme is developed to carry out multi-way branches, especially those derived from applying the SV transformation, in an efficient way.

The discussion of this paper is conducted through first presenting the SV transformation in Section 2. Then, in Section 3, the proposed multi-way scheme is described. Section 4 evaluates the effectiveness of the SV transformation and the proposed multi-way branch mechanism. Section 5 concludes the discussion of this paper.

## 2. The SV transformation

The SV transformation works by invoking shadow variables to transform program segments in which multi-way branches are originally not applicable into ones in which multi-way branches are applicable. In 2.1, two examples are used to illustrate

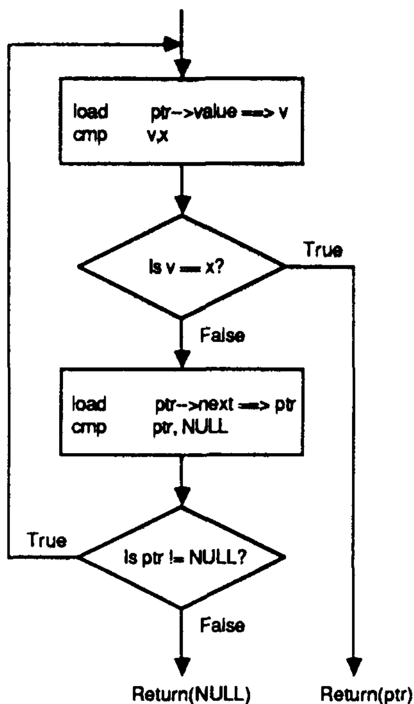



Fig. 1. Flow diagram of the core loop for searching in a linear list.

how the SV transformation works. Then, in 2.2, the procedural description of the SV transformation is presented.

### 2.1. Examples of the SV transformation

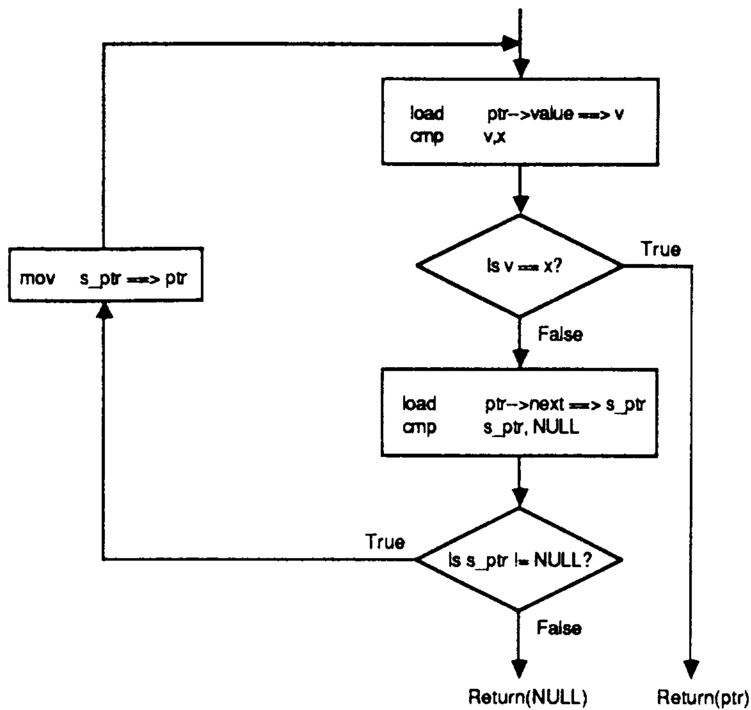
The first example used is searching in a linear list. *Figure 1* shows a typical flow diagram of the core loop for searching in a linear list. In this diagram, the embedded control dependence and data dependence rule out the possibility of parallel instruction execution as well as merging the two conditional branches to form a multi-way branch. The major blockade is the control dependence imposed by branch 'Is v == x?' on instruction 'load ptr->next ==> ptr'. The control dependence dictates that the load instruction must be scheduled after the conditional branch instruction since the value of 'ptr' must be preserved in case execution proceeds to another path of the conditional branch.

The SV transformation handles this case by introducing a shadow variable 's_ptr' and substituting instructions 'load ptr->next ==> ptr' and 'cmp ptr,NULL' with 'load ptr->next ==> s_ptr' and 'cmp s_ptr,NULL', respectively. The substitution makes these two instructions no longer cause irreversible side-effect to another path branching from 'Is v == x?'. *Figure 2(a)* shows the flow diagram after the shadow variable is introduced. In *Fig. 2(a)*, a compensation code is added at the end of the loop to copy the value of 's_ptr' into 'ptr'.

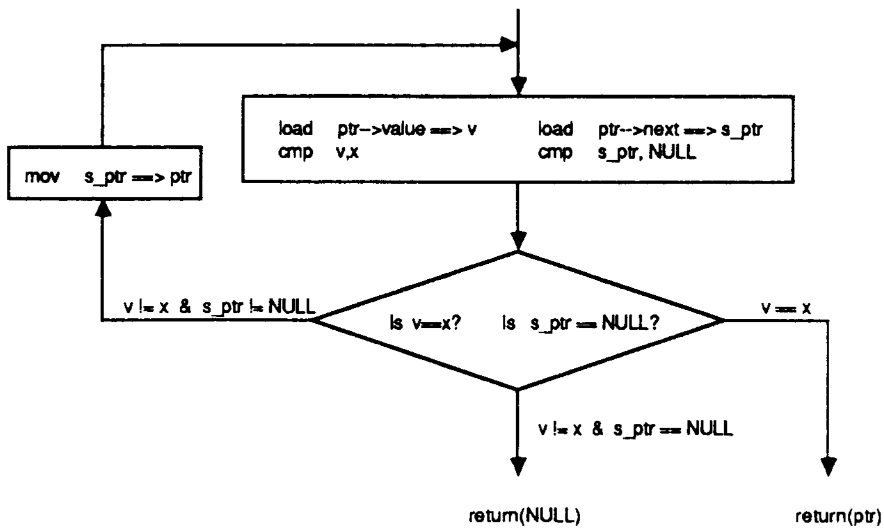
With the flow diagram in *Fig. 2(a)*, the two conditional branches can be merged to form a multi-way branch by moving instructions 'load ptr->next ==> s_ptr' and 'cmp s_ptr,NULL' up ahead of conditional branch 'Is v == x?'. *Figure 2(b)* shows the flow diagram after the merge is done.

The benefits gained by performing the SV transformation are manifest. First, the loop now contains only one branch operation instead of two. Second, instructions now can be scheduled for parallel execution in order to speedup the loop.

The first example is a fairly simple one. Let us use a more complicated case for illustration. The second example is searching in a binary tree. *Figure 3* shows a typical flow diagram of the core loop for




(a) The intermediate stage




(b) The final loop structure

Fig. 2. Loop structures of the linear search example after the SV transformation is applied.

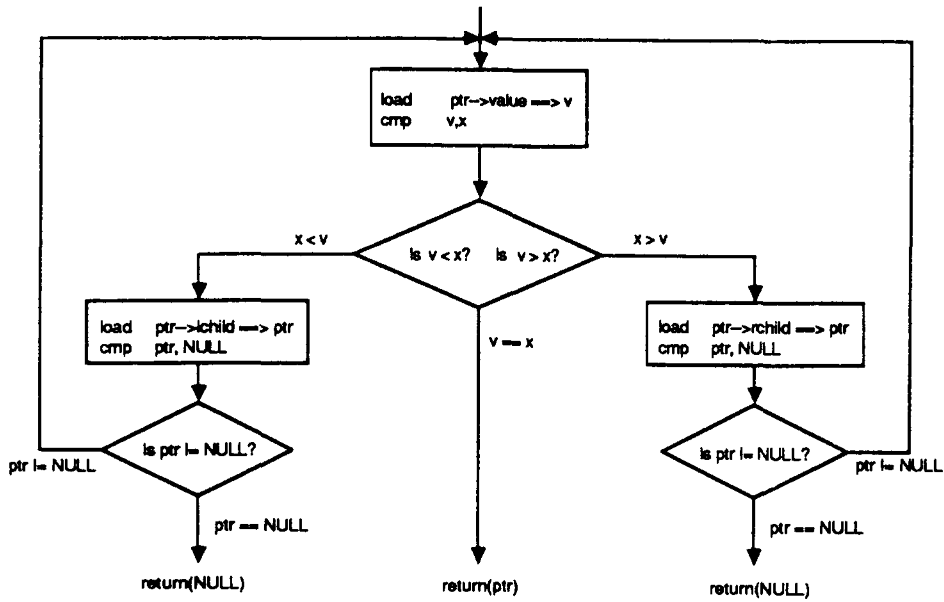



Fig. 3. Flow diagram of the core loop for searching in a binary tree.

searching in a binary tree. In this diagram, there is already a multi-way branch, 'Is v < x? Is v > x?', but further merging of branch operations is not possible. The major blockade is the control dependence imposed by multi-way branch 'Is v < x? Is v > x?' on instructions 'load ptr —> lchild ==> ptr' and 'load ptr —> rchild ==> ptr'.

The SV transformation handles this case by introducing shadow variables 's_lchild' and 's_rchild' and making the following substitutions:

- substitute 'load ptr —> lchild ==> ptr' and 'cmp ptr,NULL' on the left path branching from 'Is v < x? Is v> x?' with 'load ptr —> lchild ==> s_lchild' and 'cmp s_lchild,NULL', respectively.

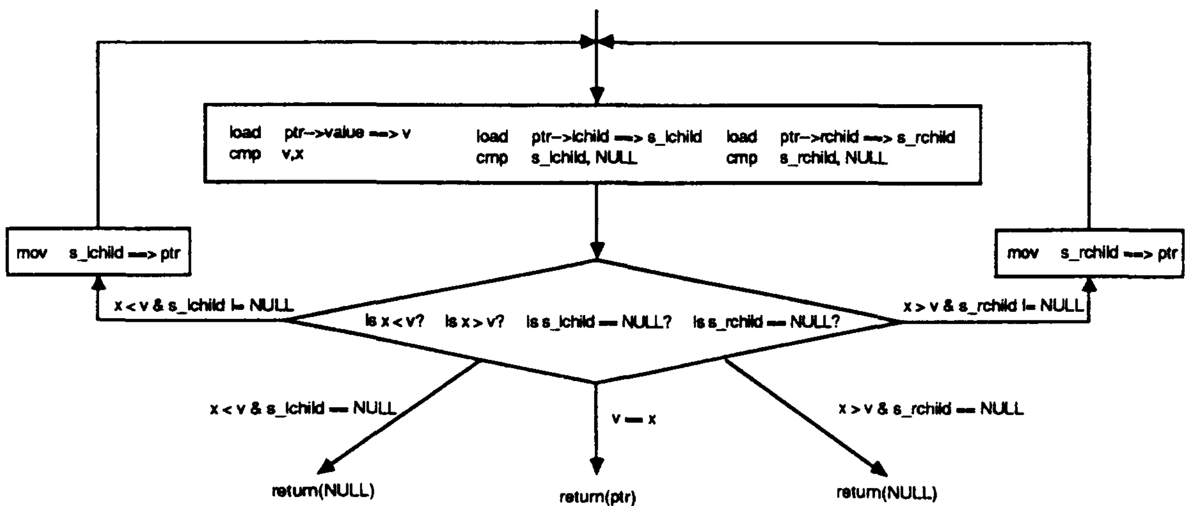



Fig. 4. Loop structure of the binary search example after the SV transformation is applied.

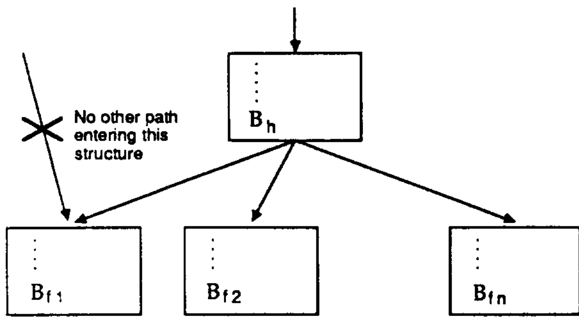



Fig. 5. Structure of program segments eligible for applying the SV transformation on.

- substitute 'load ptr—>rchild ==> ptr' and 'cmp ptr, NULL' on the right path branching from 'Is v < x? Is v > x?' with 'load ptr—>rchild ==> s_rchild' and 'cmp s_rchild, NULL', respectively.

The substitutions make the instructions involved no longer cause irreversible side-effect to the other paths branching from 'Is v < x? Is v > x?' and, therefore, can be moved up across 'Is v < x? Is v > x?'. *Figure 4* shows the flow diagram after the SV transformation is carried out.

*2.2. Procedural description of the SV transformation*

The procedural description of the SV transformation is presented in the following:

(1) Identify a group of branch operations as the target. The group of branch operations should contain a branch operation as the header, denoted by $B_h$, with all other branch operations, denoted by $B_{f1}, B_{f2}, ..., B_{fn}$, located on the immediately following basic blocks branching from $B_h$. For the time being, we only consider the cases in which the basic block containing $B_h$ is the only entry to the whole structure. *Figure 5* depicts the structure of the program segments eligible for applying the SV transformation on. In *Fig. 5*, each box represents a basic block.

(2) For branch $B_{fi}$, $1 <= i <= n$, if there is no instruction on the path from $B_h$ to $B_{fi}$ other than $B_h$ and $B_{fi}$, then $B_{fi}$ can be merged with $B_h$ without further actions.

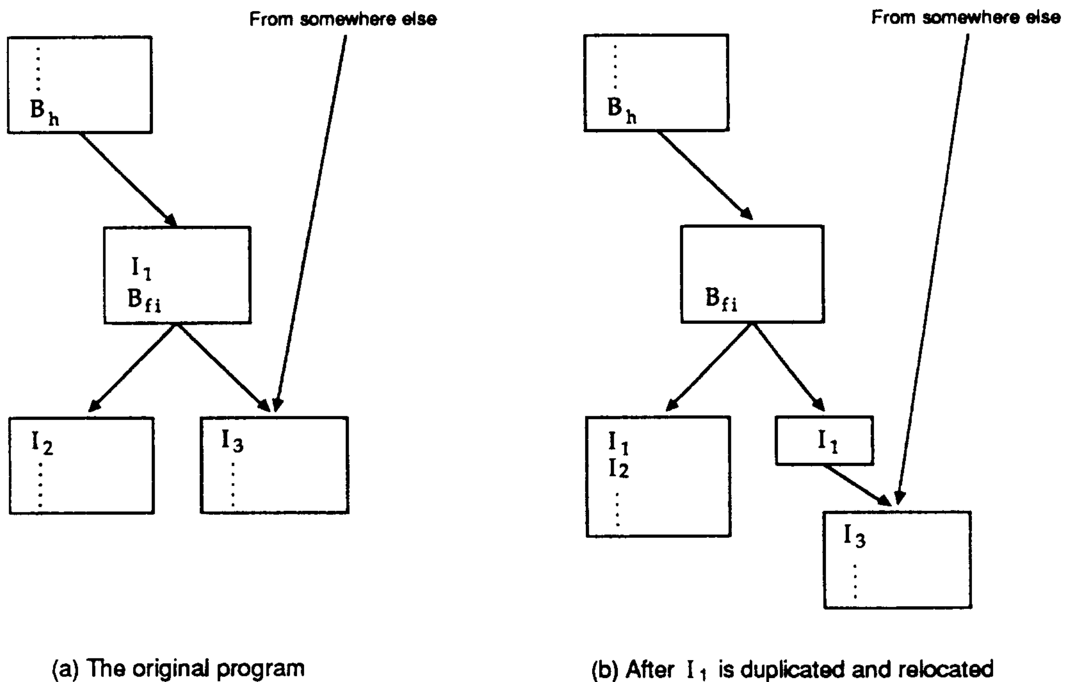



Fig. 6. Example of code duplication and relocation.

(3) If there are instructions on the path from $B_h$ to $B_{fi}$ other than $B_h$ and $B_{fi}$ and $B_{fi}$ is not data-dependent on any of these instructions, then these instructions can be relocated into the paths branching from $B_{fi}$. The relocation of instructions generally invokes code duplication. *Figure 6* demonstrates the action of code duplication and relocation performed for this case. Again, in *Fig. 6*, each box represents a basic block. Once the code duplication and relocation is done, $B_{fi}$ can be merged with $B_h$ just like the case in Step 2.

(4) If $B_{fi}$ is data-dependent on some of the instructions, $I_1, I_2, ..., I_k$, on the path from $B_h$ to $B_{fi}$, then this is a case where shadow variables need to be introduced. Before we introduce shadow variables, we first invoke the action in Step 3 above to relocate those instructions on the path from $B_h$ to $B_{fi}$ that $B_{fi}$ is not data-dependent on. These instructions can be moved into the paths branching from $B_{fi}$. Then, we introduce a shadow variable for each of the variables that $I_1, I_2, ..., I_k$ write to and make necessary substitutions. The substitution of shadow variables will transform $I_1, I_2, ..., I_k$ into a new group of instructions, denoted by $I'_1, I'_2, ..., I'_k$, that no longer cause irreversible side-effects to the other paths branching from $B_h$. As a result, $I'_1, I'_2, ..., I'_k$ can be moved up across $B_h$ safely. With $I'_1, I'_2, ..., I'_k$ relocated, the path from $B_h$ to $B_{fi}$ now contains no instructions and $B_{fi}$, with necessary substitutions of shadow variables, can be merged with $B_h$ just like the case in Step 2. Certainly, due to the introduction of shadow variables, compensation codes must be added to copy the values of the shadow variables into their corresponding authentic variables. The compensation codes can be added in the paths branching from $B_{fi}$. In the discussion above, we assume that the superscalar processor features the capability to handle false traps caused by speculative execution [4,5]. If it is not the case, then instructions $I_1, I_2, ..., I_k$ must not invoke memory reference or division operations for the SV transformation to be applied. Otherwise, the memory reference and division opera-

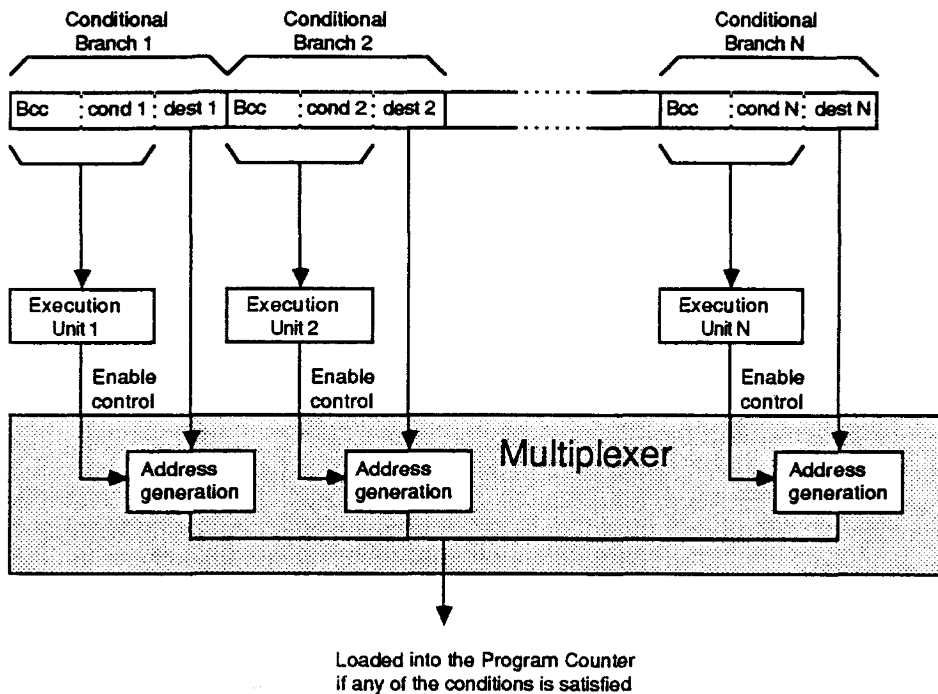



Fig. 7. Operation of the proposed multi-way branching mechanism.

tions in $I_1$, $I_2$, ..., $I_k$ could cause false fatal faults [4,5].

## 3. The proposed multi-way branch mechanism

The proposed multi-way branch mechanism is developed in conjunction with the SV transformation. The multi-way branching mechanism is developed to carry out multi-way branches, especially those that are derived from applying the SV transformation, efficiently. The proposed multi-way branching mechanism operates by executing multiple conditional branch instructions concurrently. The parallel execution of multiple branch instructions involves first dispatching each of the branch instructions that constitute the multi-way branch instance to the corresponding execution unit for concurrent evaluation of the branch conditions. Then, the destination address of the branch instruction whose condition is satisfied is selected and loaded into the program counter. In the case where no condition is satisfied, execution proceeds to next instruction as normal. Here, it is assumed that no more than one condition is satisfied at the same time. This assumption imposes a rule that the compiler or the programmer must follow. The compiler or the programmer can only schedule conditional branch instructions that have mutually-exclusive branch conditions for concurrent execution. *Figure 7* illustrates the operation of the proposed multi-way branching mechanism.

Table1
Options for utilizing the delayed slots of multi-way branches

| Option 1 | Option 2 | Option 3 |
|---|---|---|
| Execute the instructions in the delay slot before taking the branch action | Before taking the branch action, execute the instruction in the delay slot that is to be executed by the same execution unit as the conditional branch instruction whose action is taken and nullify the rest of the instructions in the delay slot | Nullify the instructions in the delay slot |

## 4. Evaluation of effectiveness

This section evaluates the effectiveness of the SV transformation and the proposed multi-way branch mechanism. The evaluation is conducted through measuring the performance of a superscalar processor model that features the proposed multi-way branch mechanism. Other features of the superscalar processor model include:

(1) An extended instruction set that is derived from the Sparc processor [7].
(2) Static scheduling [6].
(3) A 5-stage pipeline with single-cycle delay for load and branch operations. The 5 pipeline stages are, in their execution order: (1) Instruction fetch, (2) Instruction decode and register

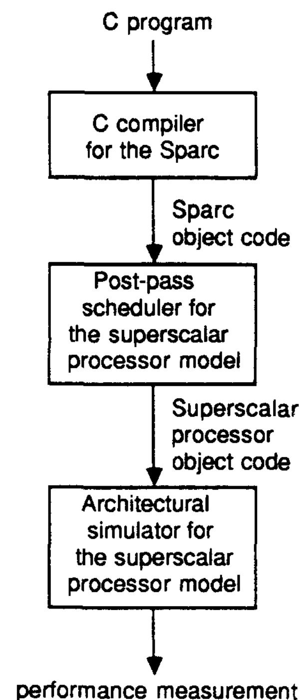



Fig. 8. Simulation environment for measuring the effectiveness.

fetch, (3) ALU operation, (4) Memory operation, (5) Register write back.
(4) Three options for utilizing the delay slots of multi-way branches as illustrated in *Table 1*.

*Figure 8* illustrates the simulation environment set up for measuring the performance of the superscalar processor model. First, the object code generated by a C compiler for the Sparc microprocessor [7] is fed into a post-pass scheduler. The post-pass scheduler performs instruction scheduling on the Sparc object file and outputs an executable file for the superscalar processor model. The executable file along with input data, are then fed into an architecture simulator that emulates the operation of the superscalar processor model. The execution time taken by the superscalar processor model is measured by counting the number of clock cycles the simulator takes to complete the task.

*Table 2* lists the speedup of the superscalar processor model over a conceptual scalar RISC processor that is derived from reducing the superscalar processor model to one execution unit. In *Table 2*, the performance of the scalar RISC processor is normalized to 1. The effectiveness of the SV transformation and the proposed multi-way branch mechanism is demonstrated by the improvement from the first two columns of the data to the third and fourth columns. The speedup data in the first two columns are the performance of the superscalar processor model without exploiting the SV transformation and the proposed multi-way branch mechanism. The speedup data in the third and fourth columns are the performance of the superscalar processor model when the SV transformation and the proposed multi-way branch mechanism are applied. *Table 2* shows that SV transformation and the proposed multi-way branch mechanism contribute to an increase of performance by 11% to 14%.

## 5. Conclusion

In this paper, we presented a compiler technique called SV transformation and a multi-way branch mechanism for exploiting multi-way branches in superscalar instruction scheduling. The SV transformation can transform program segments in which multi-way branches are not applicable into ones in which multi-way branch are applicable. The proposed multi-way branch scheme is able to carry out multi-way branches efficiently, especially for those derived from applying the SV transformation. The simulation-based effectiveness evaluation conducted in this paper shows the SV transformation and the proposed multi-way branch mechanism together can boost superscalar processor performance by 11–14%.

Table 2
Speedup of the superscalar processor model over the conceptual scalar RISC processor

| Benchmark programs | Speedups without SV trans. & multi-way branches | | Speedups with SV trans. & multi-way branches | |
|---|---|---|---|---|
| | With 2 parallel execution units | With 4 parallel execution units | With 2 parallel execution units | With 4 parallel execution units |
| Shortest path | 1.36 | 1.37 | 1.75 | 1.81 |
| Binary search | 1.31 | 1.32 | 1.44 | 1.56 |
| Binary tree traverse | 1.27 | 1.29 | 1.35 | 1.38 |
| Bucket sort | 1.50 | 1.54 | 1.54 | 1.61 |
| Fibonacci number | 1.80 | 1.85 | 1.85 | 1.92 |
| GCD | 1.15 | 1.16 | 1.54 | 1.55 |
| Find the minimum | 1.26 | 1.26 | 1.51 | 1.66 |
| Merge sort | 1.60 | 1.67 | 1.80 | 2.00 |
| Pattern match | 1.48 | 1.58 | 1.52 | 1.63 |
| String compare | 1.35 | 1.50 | 1.38 | 1.54 |
| Average | 1.41 | 1.46 | 1.57 | 1.67 |

## References

[1] M. Johnson, *Superscalar Microprocessor Design* (Prentice-Hall, Englewood Cliffs, NJ, 1991).

[2] N.P. Jouppi, The nonuniform distribution of instruction-level and machine parallelism and its effect on performance, *IEEE Trans. Comput.* 38, (12) (Dec. 1989).

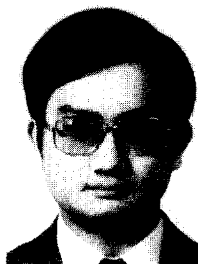
[3] M.D. Smith, M. Johnson and M.A. Horowitz, Limits on multiple instruction issue, *Proc. 3rd Internat. Conf. on Architectural Support for Programming Languages and Operating System* (1989).

[4] P.,P. Chang, S.A. Mahlke, W.Y. Chen, N.J. Warter and W.M. Hwu, IMPACT: An architectural framework for multiple-instruction-issue processors, *Proc. 18th Annual Internat. Symp. on Computer Architecture* (May 1991).

[5] M.D. Smith, M. Horowitz and M.S. Lam, Efficient superscalar performance through boosting, *Proc. 5th Internat. Conf. on Architectural Support for Programming Languages and Operating Systems* (1992).

[6] M.S. Lam, Instruction scheduling for superscalar architectures, *Ann. Rev. Comput. Sci.* (1990).

[7] Sun Microsystems Inc., *The Sparc Architecture Manual Version 7* (Sun Microsystems Inc., 1987).



**Yen-Jen Oyang** received the B.S. degree in information Engineering from National Taiwan University in 1982, the M.S. degree in Computer Science from California Institute of Technology in 1984, and the Ph.D. degree in Electrical Engineering from Stanford University in 1988. He is currently an Associate Professor in the Department of Computer Science and Information Engineering, National Taiwan University. His research interests include computer architecture and VLSI system design.