# Conditional Annulling for Superscalar Processors

Chun-Hung Wen, Chih-Yuan Cheng, Yun-Yu Kung, Chung-Yu Chang,
and Yen-Jen Oyang

*Department of Computer Science and Information Engineering*

*National Taiwan University*

*Taipei, Taiwan* , China

## Abstract

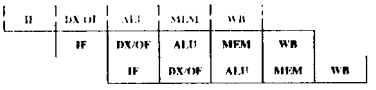
This paper proposes an architectural feature for superscalar processors called conditional annulling. The conditional annulling mechanism achieves both the effects of conditional execution and speculative execution without incurring a significant increase of hardware complexity and cost. An effectiveness evaluation carried out in this paper shows the conditional annulling mechanism is effective in improving superscalar processor instruction scheduling.

**Keywords:** superscalar processor, pipeline, conditional execution, speculative execution, conditional annulling.

## 1 Introduction

It is well known that 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 [1, 2]. Branch instructions cause two negative effects to superscalar processors. The first effect is the incurring of control dependence in the program which, in turns, seriously limits the instruction-level parallelism exploitable. 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.

Motivated by the observations discussed above, we developed a mechanism called conditional annulling for superscalar processors. The conditional annulling mechanism achieves both the effects of conditional execution [3] and speculative execution [6, 7, 8]. In some instances, conditional annulling can be applied to eliminate branch operations when dealing with branch paths of program flow. In some other instances, conditional annulling facilitates instruction scheduling across basic block boundaries




| | |
|---|---|
| **IF** | Instruction fetch. |
| **DX/OF** | Instruction decode, operand fetch. |
| **ALU** | Arithmetic/Logic operation and setting of condition flag contents. |
| **MEM** | Memory access, perform memory read/write. |
| **WB** | Write back, update register file. |

Figure 1: The pipeline of our baseline machine model

by allowing speculative instruction execution to a limited extent. The main advantage of the conditional annulling mechanism is that it does not incur a significant increase of hardware complexity and cost. It needs no shadow registers as required for performing general speculative execution. The key point behind the conditional annulling mechanism is effective exploitation of the pipeline structure of modern microprocessors. If compared with guarded execution [4], or called predicated execution [5], which achieves the effect of conditional execution, the conditional annulling offers more instruction scheduling flexibility by providing limited speculative execution.

In the rest part of this paper, we will describe the basic mechanism of conditional annulling in section 2. In section 3, we will elaborate how conditional annulling can be applied in compiling programs for superscalar processors. An effectiveness evaluation of the conditional annulling mechanism is presented in section 4.

## 2 The Conditional Annulling Mechanism

To facilitate the discussion of the conditional annulling mechanism, let us first build a superscalar processor model that servers as our baseline machine. The superscalar pro-

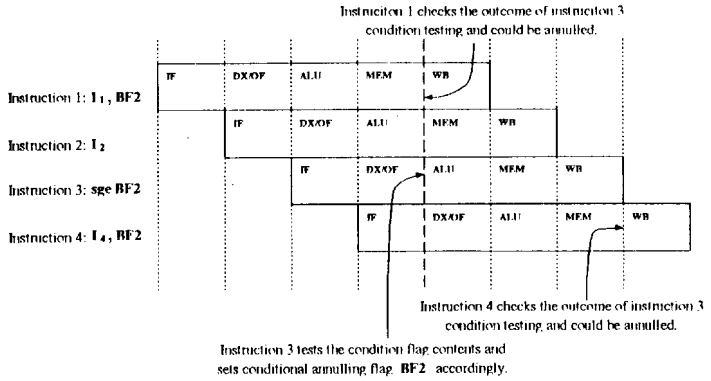



Figure 2: An example of conditional annulling

cessor model executes instructions in a RISC-style 5-stage pipeline [9] as illustrated in Fig. 1. It has one set of condition flags that are set according to the outcome of arithmetic and logical operations. On executing a branch instruction, the processor tests the contents of condition flags and decides the action to take in the second pipeline cycle. Since the testing of the condition flag contents is carried out in the second pipeline cycle and the output of an instruction is not written into the register file until the fifth pipeline cycle, the execution of an instruction can be conditionally annulled based on the outcome of a condition testing carried out by an instruction that is issued two clock cycles later. Based on this observation, we introduce a class of instructions that test condition flag contents in the second pipeline cycle and store the outcome in one of the boolean flags called Boolean Flags, **BF**s. Fig. 2 illustrates the operation of conditional annulling. In Fig. 2, instruction 3, `sge BF2`, tests the condition flag contents and stores the outcome in conditional annulling flag `BF2`. The status of `BF2`, then, determines if instructions 1 and 4 should be annulled when the execution proceeds to the fifth pipeline cycle. In practice, we can conclude that an instruction which tests the condition flag contents is able to conditionally annul any instruction that is issued no earlier than 2 cycles before it.

The discussion so far is for general cases. There are some other cases to consider. One case is the store instruction. A store instruction must be annulled at the beginning of the fourth pipeline cycle, instead of the fifth cycle, for conditional annulling to take effect because the output data is written into the memory in the fourth cycle of the pipeline. Another case is the arithmetic and logical instructions that set the condition flags. Since setting of condition flags is done in the third pipeline cycle (the ALU cycle), see Fig. 1, it is impossible for an instruction that is issued later to annul the effect of condition setting.

What we presented in this section is the basic mechanism of conditional annulling. In the next section, we will elaborate how to apply this technique in generating codes for superscalar processors.

| | |
|---|---|
| **DX/OF** | Instruction Decode and Operand Fetch Cycle: For instructions that test condition flag contents, e.g. conditional branch and `sge` instruction in Fig. 2, the outcome is recorded in a **BF**. |
| **MEM** | The Memory Access Cycle: For a store instruction, the status of the specified **BF** is examined to determined if the instruction is to be annulled. |
| **WB** | The Write Back Cycle: The status of the **BF** is examined to determine if the register file is to be updated. |

Figure 3: The modifications of pipeline structure for the demonstration superscalar processor.

## 3 Code Optimizations with Conditional Annulling

It is necessary to define a demonstration superscalar processor with conditional annulling mechanism for further discussion. Except instruction annulling operations, and several **BF**s, the demonstration processor inherits all the architectural features of the baseline processor defined in Section 2. Fig. 3 shows the operations that the demonstration processor needs to carry out in the pipeline cycles in addition to the operations in Fig. 1.

To transform a program targeted for the baseline processor to the demonstration processor, the code optimizer

simply chooses the instructions that do not violate data dependencies and then moves them upward across the branch instruction within two clock cycles. By doing this, the demonstration processor can achieve both the effects of conditional execution and speculative execution. Such code movement is straightforward and easy to implement. Following examples illustrate this basic optimization technique.

The first example used to illustrates the application of conditional annulling is the conditional expression construct in C programs. The expression `max = (a > b)?a : b;` assigns the larger number between `a` and `b` to `max`. For the baseline processor, this expression can be realized with the following code segment:

```
    (I1) cmp a,b;   (I2) mov a,max;
    (I3) ble,a L1;
    (I4) mov b,max;
L1:
```

In this code segment, `I1` and `I2` are executed in the same clock cycle and I4 is in the delayed slot of conditional branch`I3`. This code segment takes three clock to complete.

With conditional execution, the conditional branch can be eliminated as shown below:

```
    (I1) cmp a,b;   (I2) mov a,max;
    (I5) sle R4;
    (I6) mov b,max,R4;
```

In this code segment, condition setting instruction `I5` sets the condition in `R4` whose content then determine whether instruction `I6` should be executed. Though the branch instruction is eliminated, this code segment still takes three clock cycles to complete.

With conditional annulling, the code segment can be further compacted. The code listed below takes only two clock cycles to complete.

```
    (I1) cmp a,b;   (I2) mov a,max;
    (I7) sle BF2;   (I8) mov b,max,BF2;
```

The key operation here is the moving of instruction `I8` one clock cycle ahead. Such new freedom of code movement provides compilers an opportunity of performing a more optimized code scheduling.

The second example shows speculative execution with conditional annulling. The C program:

```
for (i=head; i; i=i->next) sum
+= i->data;
```

sums the values in the `data` field of the linked list. With conditional annulling, the program segment can be realized by the code segment shown in Fig. 4. In this code segment, instruction `I10` speculatively preload `data` of the first iteration. Also, instruction `I17`, which is in the loop body, carries out the speculative data loading for the succeeding iterations. The execution of these two instructions is subject to possible annulling depending on the testing result of instruction `I14`. Since instructions `I10` or `I14` may cause memory violation fault if they are to be annulled, the fault needs to be suppressed until the condition testing is settled.

| name | -uni | -Osched | -Oreg | -O |
|---|---|---|---|---|
| binary | 1.000 | | 1.515 | 2.290 |
| bubble | 1.000 | | 1.837 | 3.262 |
| fib | 1.000 | | 1.885 | 1.974 |
| gcd | 1.000 | | 1.863 | 2.972 |
| pat | 1.000 | | 1.850 | 2.630 |
| perm | 1.000 | | 2.360 | 2.528 |
| qsort | 1.000 | | 1.835 | 2.296 |
| sieve | 1.000 | | 1.897 | 3.895 |
| Average | | | 1.880 | 2.371 |

Table 1: Performance benchmark data

# 4 Evaluation of the Effectiveness of Conditional Annulling

To evaluate the effectiveness of the conditional annulling mechanism, we use the demonstration superscalar processor model as the target machine. The benchmark process is conducted through first feeding the benchmark program into a compiler that generates the object code for our demonstration superscalar processor. Then, the object code, along with a set of input data, are fed into an architectural simulator that emulates the operation of our demonstration machine to count the execution time of the benchmark program. Table 1 shows the execution time data collected in benchmark runs. A description of the benchmark programs is given in Table 2. We expect the conditional annulling mechanism will be most effective in scheduling instructions within inner loops. Therefore, in our evaluation, we selected mainly programs with such characteristics.

The first column of data in Table 1 is the reference of the speedup measurement. It is the execution time of our demonstration superscalar processor when it is reduced to issue one instruction per clock cycle. The second column of data is the speedup of our demonstration superscalar processor if the conditional annulling mechanism is not turned on. Though conditional annulling is turned off in this case, the compiler still performs global register allocation, list scheduling within basic blocks, and constant propagation within basic blocks. The third column of data is the speedup of our demonstration superscalar processor if the conditional annulling mechanism is turned on. From the data in columns 2 and 3, we can conclude that conditional annulling is an effective mechanism to enhance superscalar processor processing power.

```
     (I9)   mov head,R1;
     (I10)  ld [R1+data],R2,BF2;   (I11)   jump L1;
     (I12)  cmp R1,0;   /* delay slot of jump */
L2:  (I13)  cmp R1,0;
L1:  (I14)  bne L2,BF2;   (I15) ld [R1+next],R1,BF2;  (I16)  add sum,R2,sum,BF2;
     (I17)  ld [R1+data],R2,BF2;    /* delay slot of bne */
```

Figure 4: Example of speculative data preloading

| Benchmark | Description |
|---|---|
| binary | binary search of an item in an array. |
| bubble | bubble sort of an integer array. |
| fib | enumeration of Fibonacci sequence by recursive calls. |
| gcd | finding greatest common divider of two numbers. |
| pat | pattern matching in an integer array. |
| perm | enumeration of all permutations of an array by recursive calls. |
| qsort | quick sort of an integer array. |
| sieve | sieve of Eratosthenes |

Table 2: Description of the benchmark programs

## 5 Conclusion

In this paper, we proposed the conditional annulling mechanism for superscalar processors. The conditional annulling mechanism achieves both the effects of conditional execution and speculative execution and requires no sophisticated hardware for implementation. An effectiveness evaluation carried out in this paper shows the conditional annulling mechanism is effective in improving superscalar processor instruction scheduling.

## References


[1] Jouppi, N.P., " The Nonuniform Distribution of Instruction-level and Machine Parallelism and its Effect on Performance", IEEE Trans. on Computer, Vol.38, No.12, Dec, 1989.

[2] Smith, M.D., Johnson, M., and Horowitz, M.A., "Limits on Multiple Instruction Issue", Proc. of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems, 1989.

[3] Adams, R.G., Gray, S.M., and Steven, G.B, "Utilizing Low Level Parallelism in General Purpose Code: the HARP Project", Microprocessing and Microprogramming, No. 29, 1990.

[4] Peter Y.T. Hsu, and Edward S. Davidson, "Highly Concurrent Scalar Processing", Proc. of the 13th Annual International Symposium on Computer Architecture, 1986.

[5] B. Ramakrishna Rau, David W.L. Yen, Wei Yen, and Ross A. Towle,. "The Cydra 5 Departmental Supercomputer: Design Philosophies, Decisions, and Trade-offes" IEEE Computer, January, 1989.

[6] Chang, P.P., Mahlke, S.A., Chen, W.Y., Warter, N.J., and Hwu, W.M., "Impact: An architectural Framework for Multiple-Instruction-Issue Processors", Proc. of the 18th Annual International Symposium on Computer Architecture, May, 1991.

[7] Smith, M.D., Lam, M.S., and Horowitz, M.A., "Boosting Beyond Static Scheduling in a Superscalar Processor", Proc. of the 17th Annual International Symposium on Computer Architecture, 1990.

[8] Johnson, M., *Superscalar Microprocessor Design*, Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[9] Hennessy, J.L., and Patterson, D.A, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers Inc., 1990.