DIVISION OF COMPUTER SCIENCE

An Explicitly Declared Delayed-Branch Mechanism for a Superscalar Architecture

Roger Collins
Gordon Steven

Technical Report No.197

May 1994

An Explicitly Declared Delayed-Branch Mechanism for a Superscalar Architecture

Roger Collins
comrrc@herts.ac.uk

Gordon Steven
comqgbs@herts.ac.uk

Department of Computer Science, University of Hertfordshire
College Lane, Hatfield, Herts. AL10 9AB

One of the main obstacles to exploiting the fine-grained parallelism that is available in general-purpose code is the frequency of branches that cause unpredictable changes in the control flow of a program at run-time. Whenever a branch is taken, a performance penalty may be incurred as the processor waits for instructions to be fetched from the branch target stream. RISC processors introduce a delayed-branch mechanism which defines branch delay slots into which code can be scheduled. This strategy allows the processor to be kept busy executing useful instructions while the change of control flow takes place. While the concept of delayed-branches can be readily extended to VLIW architectures, it is less clear how it should be incorporated in a superscalar architecture. This paper proposes a general branch-delay mechanism which is suitable for a range of code-compatible superscalar processors and which completely avoids the need to introduce NOPs into the code. This technique was developed as an integral part of the HSP superscalar project. HSP is a superscalar architecture currently being developed at the University of Hertfordshire with the aim of using compile-time instruction scheduling to achieve an order of magnitude speed-up over traditional RISC architectures for a suite of non-numeric benchmark programs.

Keywords: delayed branch; superscalar; instruction-level parallelism; code scheduling; conditional branches.

1. Introduction

Whenever a branch instruction evaluates as taken, the latency associated with accessing instructions from the branch-target stream results in at least one wasted processor cycle, known as the branch penalty. RISC architectures make use of a delayed branch mechanism to overcome this problem by giving the processor early notice of any possible changes in control flow. The branch is moved up in a basic block of code to allow some of the instructions following the branch to be executed during the branch penalty cycles. Each branch, in effect, has branch-delay slots appended to it into which the compiler can schedule useful instructions.

This delayed-branch mechanism can be extended to VLIW architectures, where each branch-delay slot is enlarged to hold multiple short instructions that can be executed concurrently. There are some serious disadvantages in the VLIW approach as unused portions of all long instruction words must be filled with 'NOP' instructions if insufficient short instructions can be found. This process can result in a significant expansion in static code size. Also, code that has been compiled for a particular VLIW processor must be re-compiled if it is to run on a different member of the processor family. This lack of code compatibility across a range of implementations is seen as a serious disadvantage by machine users.

Superscalar processors take in sequential program code where there is no explicit concept of parallel instruction groups and rely instead on hardware for dynamic instruction scheduling. To solve the branch penalty problem, superscalars often use run-time branch prediction. If delayed branches are to be extended to superscalars, some mechanism must be found that allows the scheduler to indicate to the processor which instructions have been moved to fill the conceptual branch delay slots.

2. The HSP Architecture

HSP (Hatfield Superscalar Processor) is a new architecture currently being developed at the University of Hertfordshire. Our intention is to speed up program execution by using an Instruction Scheduler to expose the instruction-level parallelism at compile-time, rather than dynamic scheduling with branch prediction at run-time. This strategy removes the need for out-of-order issuing of instructions or dynamic register renaming in the HSP processor. The HSP processor can therefore be thought of as a "minimal" superscalar design with simplified hardware to implement strict in-order issuing of instructions. The HSP architecture supports the use of conditional execution where multiple "guard" Boolean conditions can be attached to any instruction. Such instructions are only executed if their Boolean guards evaluate to "True" at run-time.

The HSP architecture fetches instructions from memory into an Instruction Buffer. Parallel groups of instructions are then issued from the Instruction Buffer, via Instruction Decode Units, to the processor's Functional Units, subject to data

dependencies and resource limitations. Individual instructions are sent to one of a common pool of Functional Units that is capable of processing the instruction. The Functional Units then execute instructions and write back result values to the appropriate registers. Branch instructions are processed by special Branch Units that can flush instructions from the Buffer if a branch is taken.

3. The HSP Branch Instruction

The HSP architecture provides a generalised delayed-branch mechanism [1][2] where a count value is encoded directly into each branch instruction. This count value specifies the number of instructions, following the branch instruction, that must always be dispatched for execution, irrespective of the branch evaluation at run-time. Instructions that are placed within the scope of the branch count value are said to be in the branch's "branch-delay region", analogous to the branch delay slots used in RISC architectures. When processing a branch instruction, instructions from the branch target stream are not fetched from memory until all the branch-delay code is in the Instruction Buffer or already being executed. Branch instructions can themselves be promoted up into the branch-delay region of an earlier branch, allowing a processor with several Branch Units to handle multiple branch instructions concurrently. Those branches that evaluate as taken are then allowed to take effect, one per machine cycle.

Initially, each branch instruction has empty instruction groups appended to it to represent the number of branch penalty cycles experienced at run-time. The HSP Instruction Scheduler then moves instruction groups into these empty branch penalty groups. The branch count value is adjusted to take account of the total number of instructions actually moved into the branch's delay region. Sufficient code must be promoted into the branch-delay region to fully occupy the processor during the branch-cycle as well as all of the branch-penalty cycles.

```
1   branch to target (7) ;
2   a := b + c;  ┐
3   d := 50;     │
4   e := 80;     │
5   f := f - a;  ├ branch delay region
6   g:= g - d;   │
7   a := e - d;  │
8   c := d + b;  ┘
9   h := a + g;
```

Figure 1a Scheduled sequential code

The example in Figure 1a shows the code used to fill the delay region of an unconditional branch that has a branch-penalty of one. The (7) shown at the end of the branch instruction indicates the size of the branch-delay region and is not a branch target address. Figure 1b shows the numbered instructions transformed into parallel execution groups inside the HSP processor at run-time. Each group is limited in size by data dependencies between instructions.

cycle	instructions dispatched	
1	1 2 3 4	branch cycle
2	5 6 7 8	penalty cycle
3	9	

Figure 1b Parallel instruction groups

4. Scheduling Code for HSP

If code compatibility is to be maintained across a range of processor implementations, no implicit assumptions can be made during scheduling as to the exact parallel capabilities of the target processors. The delayed-branch instruction with its explicit branch-delay region provides an essential mechanism that allows code to be specifically scheduled for a particular implementation, yet still be compatible with other members of the processor's family. The HSP Instruction Scheduler initially identifies groups of instructions that can be executed concurrently. Attempts are then made to increase the size of such groups by moving code up in the program order. In general, this code promotion may involve both code duplication and register renaming.

For unconditional branches, suitable code can be promoted from the branch target stream to fill the branch-delay regions. However, with unpredictable conditional branches the choice of which code stream to favour for promotion is less clear. Superscalars often use run-time branch prediction to determine which branch-path to favour but suffer a performance loss if the prediction fails. In HSP, some scheduling algorithm, possibly based on a history of branch behaviour during previous program runs, could be used to determine at compile-time which branch path to favour. If a conditional branch is thought to evaluate as taken at run-time, code from the branch target stream is promoted into the branch delay region and guarded by the same Boolean condition that is controlling evaluation of the branch itself. If the conditional branch is unlikely to evaluate as taken there is no need to promote any code at all into the delay region as the processor will be assumed to carry on fetching from the current program stream. In both cases, a mispredicted branch instruction will

cause a loss of performance at run-time. In the HSP Scheduler, code from both branch paths can be promoted into the branch-delay region by using the appropriate Boolean guards so that the processor has access to both groups of code at run-time.

The HSP Instruction Scheduler will adapt Ebcioglu's algorithm [3], developed for an unconventional processor by researchers at IBM, for the more conventional HSP architecture. The HSP algorithm targets innermost program loops for particular code optimisations, such as the software pipelining of loop bodies. Software pipelining enables code from different loop iterations to be overlapped. For simple loops such action may reduce the duration of the loop body to only one or two processor cycles. If we consider an HSP processor with a branch penalty of two cycles, the minimum length of any loop is forced to be three cycles; one for the branch instruction and two for the penalty cycles. If the natural length of a scheduled loop is only two cycles, the effect of the branch penalty is to extend the loop iteration interval unnecessarily to three cycles. In such a case the Scheduler will deliberately extend the branch delay region beyond the implementational requirements of the target processor. This extension allows a second copy of the loop body to be included within the scope of the branch. Boolean guards are appended where necessary to ensure correct loop execution. This scenario is illustrated in the example shown in Figure 2a. The symbols *f* and *g* represent any function that can be encoded into a single HSP instruction and serve to illustrate this example, rather than typify program code.

```
for (i=0; i<n; i++)
  {
   x := f (y);
   y := g (x);
  }
```

Figure 2a A simple program loop

In Figure 2a, a data dependency between the second and first statements within the body of the **for** loop prevents them from being executed in parallel. There are also data dependencies between instructions in successive loop iterations. Figure 2b shows a pseudo-code representation of the simple **for** loop of Figure 2a. Figure 3 shows code after it has been scheduled into parallel groups using a modified version of Ebcioglu's algorithm [3] that has been extended to model branch delay slots. The boundaries in Figures 3 and 4 that surround groups of instructions indicate that these instructions can be executed concurrently.

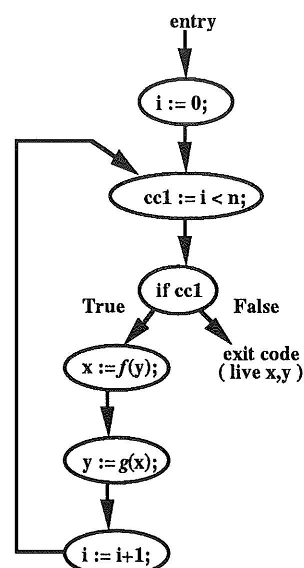


Figure 2b Control flow for simple loop

Notice that a loop prelude is generated as the loop is software-pipelined by the Instruction Scheduler.

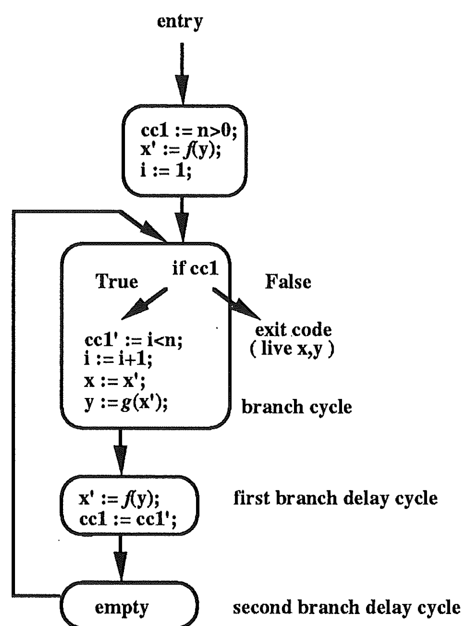
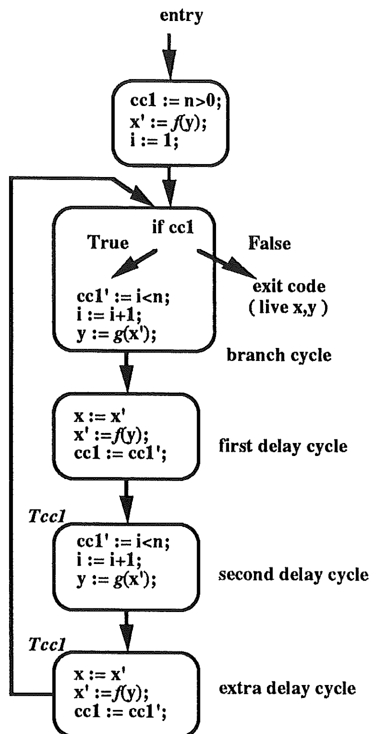


Figure 3 Branch with two penalty cycles

The HSP Scheduler has successfully found code to fill the branch cycle and to occupy the first branch delay slot. The second branch delay slot remains empty and the iteration interval for the loop is forced to be three machine cycles. If the loop body is replicated, the branch's delay region can be extended to accommodate extra code, as shown in Figure 4.



Last two groups guarded by Boolean condition Tcc1

Figure 4 Extended program loop

Execution of the extra code is guarded by a Boolean condition to ensure that the loop is exited correctly. Each iteration of the extended loop will perform two iterations of the original loop. For large values of *n*, program execution time is now proportional to *2n* , as opposed to *3n* for the code in Figure 3.

```
     entry
     1            cc1 := n>0;
     2            x' := f(y);
     3            i := 1;
loop 4            if cc1 branch to loop (12);
     5    Tcc1    cc1':= i<n;
     6    Tcc1    i := i+1;
     7    Tcc1    y := g(x');
     8    Tcc1    x := x';
     9    Tcc1    x' := f(y);
     10   Tcc1    cc1 := cc1';
     11   Tcc1    cc1':= i<n;
     12   Tcc1    i := i+1;
     13   Tcc1    y := g(x');
     14   Tcc1    x := x';
     15   Tcc1    x' := f(y);
     16   Tcc1    cc1 := cc1';
     exit code
```

Figure 5 Output code for the extended loop

The output code shown in Figure 5 has a branch count value set to twelve, with all of the loop body instructions guarded by the Boolean *True cc1*.

5. Discussion and Conclusions

The HSP delayed-branch mechanism enables code scheduling at compile-time to reduce the adverse effects of branches. The branch delay count allows code to be scheduled for a particular processor, whilst still retaining code compatibility across a range of similar processors with differing hardware resources. Code scheduled for a processor with an instruction issue rate of 8 and a branch penalty of 2 will run correctly on a more basic processor with an issue rate of 2 and a one-cycle branch penalty. Code promoted into the branch-delay region from the two branch paths is guarded by Boolean expressions to ensure that the correct code is executed at run-time. The HSP architecture allows code to be removed from the Instruction Buffer if its execution rests on a Boolean condition which is known to be false at run-time. Code in the branch-delay region that relates to the untaken branch path can be removed from the Instruction Buffer before it is selected for dispatch, thus improving the utilisation of processor resources. As the HSP Scheduler examines both branch paths, all code is considered for upward promotion in the program order, not just code from the predicted path.

The ability to arbitrarily extend the branch delay region beyond the requirements of the processor allows the Scheduler to make optimisations in special circumstances, e.g loop unrolling. Implementing the HSP delayed branch instruction involves some cost in terms of the bits required to encode the branch count value, but typical branch instructions normally have some spare bits available. However, without such a count mechanism we would have to assume implicitly some occupancy value and pad-out unfilled positions with NOPs, as in the VLIW architecture. It is the explicitly declared branch count that allows code to be scheduled for a specific processor, yet still gives the code compatibility across the range of processor implementations which is considered so important by computer users.

References

[1] Collins, R. *Developing a Simulator for the Hatfield Superscalar Processor* Division of Computer Science, University of Hertfordshire, December 1993.

[2] Steven, G. B. *The Hatfield Superscalar Architecture* Division of Computer Science, University of Hertfordshire, 1994.

[3] Moon, S. M. and Ebcioglu, K. *An Efficient Resource-Constrained Global Scheduling Technique for Superscalar and VLIW processors.* 25th Annual International Symposium on Microarchitecture, Portland, Oregon, December 1992, pp 55-71.