Delayed Branches Versus Dynamic Branch Prediction in a High-Performance Superscalar Architecture

Colin Egan, Fleur Steven and Gordon Steven
University of Hertfordshire, Hatfield, Hertfordshire AL10 9AB
Telephone: 01707 284319 Fax: 01707 284 303 email: comqgbs@herts.ac.uk

Abstract

While delayed branch mechanisms were popular with the designers of RISC processors, most superscalar processors deploy dynamic branch prediction to minimise run-time branch penalties. We propose a generalised branch delay mechanism that is more suited to superscalar processors. We then quantitatively compare the performance of our delayed branch mechanism with run-time branch prediction, in the context of a high-performance superscalar architecture that uses aggressive compile-time instruction scheduling.

keywords

superscalar, delayed branch, branch prediction, instruction scheduling

1. Introduction

Today's superscalar processors rely on aggressive pipelining and multiple instruction issue to achieve high performance. Inevitably, however, branch instructions interrupt the flow of instructions in the pipeline and degrade processor performance. RISC architectures traditionally use a delayed branch mechanism to reduce the penalty of taken branches. In contrast recent superscalar implementations have tended to abandon the RISC mechanism in favour of run-time branch prediction.

At the University of Hertfordshire, we have developed the Hatfield Superscalar Architecture (HSA), a high-performance superscalar processor model, to allow us to investigate the performance limits of compile-time instruction scheduling. Our long term objective is to achieve an order of magnitude performance improvement over traditional RISC implementations that issue only one instruction in each processor cycle. A second objective is to investigate whether aggressive compile-time scheduling can be used to simplify the hardware requirements of superscalar implementations. We were therefore attracted to the traditional RISC delayed branch mechanism by its inherent simplicity and low hardware implementation cost.

This paper describes the generalised delayed branch mechanism that we have developed for the HSA architecture, including a recent simplification of our mechanism. We then use one of our instruction schedulers to quantify the performance of delayed branches and to compare delayed branches with a more conventional dynamic branch prediction mechanism.

2. The Hatfield Superscalar Architecture

The Hatfield Superscalar Architecture (HSA) was developed as a vehicle for instruction scheduling research. It has been described as a minimal superscalar architecture [1] since it embodies a hybrid technology that combines the best features of VLIW (Very Long Instruction Word) and superscalar architectures. HSA is a load and store architecture with a RISC instruction set derived from its predecessor HARP [2]. Separate integer and Boolean register files are provided. The one bit Boolean registers are used to store branch conditions and to implement guarded instruction execution. A simple four-stage pipeline is used:

0-8186-8215-9/97 $10.00

IF	Instruction Fetch
ID	Instruction Decode
EX	Execute
WB	Write Back

During the IF stage, multiple instructions are fetched into an Instruction Buffer from the instruction cache. In the ID stage, instructions are then issued in program order from the Instruction Buffer to functional units. Conditional branches are also resolved in the ID stage.

While more details of the architecture can be found elsewhere [3], two further features are directly relevant to this paper. First, all HSA instructions, including branches, may be guarded by one or more Boolean conditions. The following divide instruction, for example, will only be executed if the value in Boolean register seven evaluates to false (zero) at run time:

```
FB7 DIV R1, R6, R13
```

Second, the processor attempts to avoid issuing instructions from the Instruction Buffer if the associated guard condition has already failed. Instructions are therefore marked as "squashed" in the Instruction Buffer if they have remained in the buffer for a full cycle without being issued and if the associated Boolean condition evaluates to false. To avoid increased pressure on the processor cycle time, the ID stage evaluates squashing conditions in parallel with its primary function of decoding and issuing instructions.

To achieve high performance, an instruction scheduler reorders the HSA assembly language code to form groups of instructions that can be issued to functional units in parallel at run time. These instruction groups are then presented to the processor as traditional sequential code. The ID stage has the task of reconstructing the original instruction groups and of issuing them to functional units for parallel execution. This involves checking that each instruction being issued does not require the result of another instruction that is being issued at the same time. Since each instruction pair must be checked for dependencies, the complexity of the dependency checking increases in proportion to the square of the issue rate and places increasing pressure on the ID stage cycle time.

In the ID stage, the processor rebuilds instruction groups that have already been assembled at compile time. We are therefore now investigating the idea of marking the end of each instruction group within the instruction stream. Only a single bit is required in each instruction to flag the end of each parallel group. Issuing instructions then simply involves scanning through the Instruction Buffer looking for the first end of group flag. No dependence checks are now required between instructions within the same group. Instruction issue is subject only to functional unit and operand availability. The instruction group flags effectively encapsulate information about compile-time instruction groups, yet do not sacrifice compatibility over a range of processor designs.

3. Delayed branch mechanism

Delayed branches first became fashionable with the introduction of RISC ideas in the early 80s. In many RISC processors, a fixed number of instructions after each branch instruction is always executed, irrespective of whether the branch is taken or not. The compiler is then given the task of placing useful instructions in the branch delay slots. Since only one branch delay slot is usually provided, straightforward instruction scheduling techniques can be used to fill the branch delay slots a very high percentage of the time. Despite this, the performance improvements achieved tended to be only marginal [4]. The main reason for this poor performance is that a branch in a simple pipeline that is not taken incurs no penalty. Introducing an unfilled delay slot or an instruction from a branch target after a not-taken branch therefore increases the execution time. Improvements through filling branch delay slots must more than compensate for such cases if a net benefit is to be realised.

Slightly better performance can be achieved if branches are able to discard instructions in their delay slots when the outcome of the branch indicates that the instructions should not be executed [4]. This additional freedom makes it easier to move instructions into delay slots. We would regard this as a specialised implementation of guarded execution, where the execution condition is moved from the instruction to the preceding branch.

In spite of disappointing benefits achieved in RISC implementations, we feel that the delayed branch mechanism is worth revisiting in the superscalar context for three reasons. First, we are attracted by the simple hardware implementation. Second, with multiple instruction issue it is possible to execute instructions from multiple successor paths until a branch is resolved. Finally, instruction buffer squashing can be deployed to reduce the resultant pressure on functional unit utilisation, although not, of course, the increased pressure on instruction cache bandwidth.

A branch delay mechanism with a fixed number of delay slots is clearly unsuitable for a superscalar

architecture. If, for example, an issue rate of eight is envisaged, sixteen delay slots would be required to cater for two delay cycles. However, in most cases such a large figure would simply require the compiler to add a significant number of NOPs after each branch.

The HSA mechanism avoids this difficulty by encoding the number of instructions placed in the branch delay slots directly in each branch instruction. The implementation of this mechanism can be considerably simplified if instruction group flags are also provided. The delay slots count then becomes a count of the number of instruction groups whose issue must be completed before the branch is taken. A two bit field can specify up to three branch delay groups and is therefore likely to be adequate.

4. Branch prediction

In contrast, most current superscalar processor implementations employ run-time branch prediction. Traditionally branch prediction involves the use of a Branch Target Cache (BTC) that predicts the outcome of branches on the basis of their past behaviour. More recently Yale Patt's group [5] and others have significantly improved the accuracy of run-time branch prediction by recording additional information about the context of each branch.

Since HSA issues instructions to functional units in program order, a BTC can be added without requiring the additional complexity of a reorder buffer (or an equivalent mechanism) to recover from mispredicted branches. All branches are resolved in the ID stage, giving adequate time for subsequent instructions to be aborted without altering the machine state.

The disadvantage of a BTC is that the time penalty increases, as a percentage, in proportion to the effective instruction issue rate. This relationship is emphasised by the method used in this paper to calculate the performance of processor models with a BTC:

Total Execution Time = Execution time with perfect branch prediction +
Branch mispredictions * Misprediction penalty

As instruction scheduling reduces the first component, the danger is that the second figure will remain largely unchanged.

For the experiments described in this paper, a separate trace-driven BTC simulator was used. The traces for both unscheduled and scheduled code were generated directly by the HSA instruction-level simulator. A fully associative BTC with 32 entries was used throughout. It was also assumed that a Return Address Stack [6] would be used to provide the appropriate return addresses for subroutine returns.

5. Instruction scheduling

Trace Scheduling [7], which is perhaps the best known instruction scheduling technique, was developed by Fisher at Yale for his VLIW architecture. However, we feel that code compatibility and code expansion problems render VLIW processors unsuitable for general-purpose computation. Fortunately, instruction scheduling techniques can be equally well applied to superscalar processors. Only slight changes are required to preserve the instruction-level semantics and to ensure that the resultant code can also be executed in the traditional sequential style.

Current scheduling developments tend to build on either Modulo Scheduling Techniques [8] or on the Enhanced Percolation Scheduling algorithm [9] developed by Kemal Ebcioglu's group at IBM. With Modulo Scheduling the main challenge is to extend the technique to loop structures of arbitrary complexity. In the case of Enhanced Percolation Scheduling, the challenge is to avoid excessive code expansion.

Our latest scheduler builds on the IBM approach and can therefore schedule loops of any complexity. However, our scheduler differs in two important respects. First, each instruction is scheduled in turn and percolated up through the code structure as far as possible. In contrast, the IBM algorithm assembles each parallel instruction group in turn, by repeatedly searching forward through the code for candidate instructions. We see the resultant repeated attempts to move each instruction forward as a disadvantage which our scheduling algorithm avoids. Second, code motion across loop back edges is restricted to avoid excessive code expansion. As in the IBM algorithm, code is systematically moved across loop back edges to overlap successive loop iterations and achieve software pipelining. However, code is only moved across a loop back edge if it can be demonstrated that the instruction being moved heads a chain of instruction dependencies that limits the iteration time of at least one path through the loop.

6. Results

Our instruction scheduler is used in this paper to evaluate the relative merits of delayed branches and dynamic branch prediction. The well-known set of Stanford benchmarks is used throughout.

Our first results compare delayed branches and a BTC in systems with an instruction issue rate of one. As a Baseline Model we execute the code produced directly by our GCC compiler on a processor with a fetch and issue rate of one. All instructions are assigned unit latencies apart from divide instructions that have a latency of sixteen and multiply instructions that have a latency of three. With a cache access time of one cycle, delayed branches incur a one cycle penalty for each taken branch, but suffer no penalty if branches are not taken. With a BTC, all mispredicted branches suffer a penalty of one.

Two sets of figures are compared with the execution times achieved by our Baseline Model. In the first case, a simple standalone instruction scheduler is used to fill branch delay slots. Traditionally, code is moved into branch delay slots from three locations, from before the branch, from the branch target and from after the branch. Our branch delay scheduler always attempts to move code from before a branch into a branch delay slot. However, to avoid degrading performance when a branch falls through, code is only moved from a branch target, if the branch is unconditional, or if the branch target is also the head of a loop. Finally, there is no point in moving instructions from after a branch into a branch delay slot, since not-taken branches incur zero time penalty. On average, filling branch delay slots improves on the performance of our Baseline Model by 8% (Fig.1).

The delayed branch mechanism is then replaced with a fully associative BTC with 32 entries and two history bits. A correctly predicted branch now suffers no performance penalty but mispredicted branches incur a penalty of one cycle. Branch prediction improves performance over the Baseline Model by 12% (Fig.1), 4% better than the delayed branch mechanism.

These experiments were repeated with the cache access times increased to two cycles. The delayed branch mechanism now allows two instruction groups to be placed in delay slots after each branch, while the mispredicted branch penalty rises to two cycles. The latency of load instructions is also increased from one to two. Filling branch delay slots yields an identical 8% improvement over the Baseline Model, while the improvement obtained using the BTC falls slightly to 11% (Fig.1).

Our second set of experiments concerns code scheduled for two multiple-instruction-issue processor models, a Standard Model and a Maximal Model. Both models have an instruction fetch rate of sixteen and can issue up to sixteen instructions from the Instruction Buffer in each cycle. The primary difference is that the Standard Model can only issue two load and two store instructions in each cycle. Since, at this stage, we are primarily interested in developing the capabilities of our scheduler, no other significant resource restrictions are placed on the two models. Again we contrast code scheduled for a delayed branch mechanism with code scheduled for use with a BTC.

Initially, cache access times were set to one cycle. First, our scheduler attempts to fill single branch delay slot with an instruction group that can be issued in parallel. In these circumstances the Standard Model achieves an average speedup of 3.11 over the Baseline Model (Fig.2). The scheduler then assumes that there are no delay slots, in effect assuming perfect branch prediction. The execution times obtained were then adjusted for BTC misses which were simulated separately. With a BTC the speedup over the Baseline Model rises to 3.63 (Fig.2).

Similarly, the Maximal Model achieves a speedup of 3.22 with delayed branches, rising to 3.90 with a BTC (Fig.2). With a cache access time of one cycle, a BTC therefore shows an overall performance advantage of around 20% over a delayed branch mechanism.

Increasing the cache access time to two cycles reduces the speedup in all cases. Average speedups of 2.66 and 2.72 are achieved using delayed branches, rising to 3.62 and 3.85 with a BTC (Fig.3). At the same time the overall performance advantage of the BTC rises from around 20% to about 40%.

7. Conclusion and discussion

In the above experiments, a BTC gives significantly higher performance than the HSA generalised branch delay mechanism. Our scheduler, in its present form, therefore fails to overcome the negative impact of branch delay slots and consequently is unable to realise the theoretical benefits of the delayed branch mechanism.

Two factors account for the marked superiority of the BTC. First, the branch prediction is remarkably successful, with an average success rate of over 99%. Second, our scheduler amplifies this already high success rate by removing a significant number of branches during scheduling (Fig.4). On average the number of branch instructions executed is reduced by over 22%, even though branch removal is entirely a side effect rather than an aim of the scheduling process. This average figure conceals very high variations between individual programs, with one program loosing just over half its branches while another looses only 2.8%.

Nonetheless, some words of caution are in order. Although the speedup currently achieved is encouraging, our scheduler is far from complete. In particular, the scheduler makes no attempt to move

branches in parallel with branches or to move branches into branch delay slots. The impact of this restriction is particularly severe when two delay slots are provided. In this case the execution time of scheduled code is almost completely dictated by the branch execution time. The next stage in the development of our scheduler will be to add this additional branch code motion. In contrast, since the success rate of the BTC already compares very favourably with two-level branch prediction techniques investigated elsewhere, the number of successful branch predictions is very unlikely to increase. We feel that the high success rate observed is almost certainly a feature of these particular benchmarks.

In summary, the preliminary speedups achieved using our latest instruction scheduler, in particular those achieved in conjunction with a BTC are highly encouraging. However, the lower speedups achieved with the HSA branch delay mechanism emphasise that significant extra development is required. In particular, to achieve higher speedups, multiple branches must be executed in parallel and branch instructions must be executed in branch delay slots.

References

1 Steven G B and Collins R A Superscalar Architecture to Exploit Instruction-Level Parallelism, *Euromicro96*, 2-5 September, Prague, 1996.

2 Steven F L, Steven G B and Wang L Using a Resource Limited Instruction Scheduler to Evaluate the iHARP Processor, *IEE Proceedings - Computers and Digital Techniques*, Vol.142, No.1, January 1995, pp 23-31.

3 Steven G B, Christianson D B, Collins R, Potter R and Steven F L A Superscalar Architecture to Exploit Instruction Level Parallelism, *Microprocessors and Microsystems*, Vol.20, No.7, March 1997, pp 391-400.

4 Hennessy J L and Patterson D A *Computer Architecture A Quantitative Approach*, Morgan Kaufman, San Francisco, 2nd edition 1996.

5 Yeh T and Patt Y N Alternative Implementations of Two-Level Adaptive Branch Prediction, *19th Annual International Symposium on Computer Architecture*, Gold Coast, Australia, May 1992, pp 124-134.

6 Kaeli D and Emma P Branch History Table Prediction of Moving Target Branches due to Subroutine Returns, *18th Annual International Symposium on Computer Architecture*, May 1991, pp34-42.

7 Fisher J A Trace Scheduling: A Technique for Global Microcode Compaction, *IEEE Transactions on Computers*, C-30, (7), July 1981, pp 478-490.

8 Rau B R and Fisher J A Instruction-Level Parallel Processing: History, Overview and Perspective, *The Journal of Supercomputing*, Vol.7, No. 1/2, 1993, pp9-50.

9 Ebcioglu K, Groves R D, Kim K, Silberman G M. and Ziv I VLIW Compilation Techniques in a Superscalar Environment, *SigPlan94*, Orlando, Florida, 1994, pp 36-48.

Fig.1 Delayed Branch versus BTC with Issue Rate of One
(Cache access times one cycle (1) and two cycles (2))

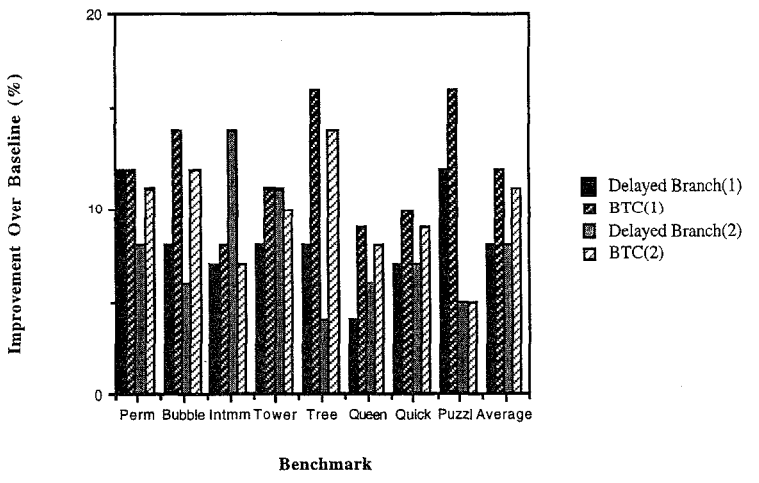

Fig.2 Delayed Branch versus BTC with Scheduled Code
(Cache access time of one cycle)

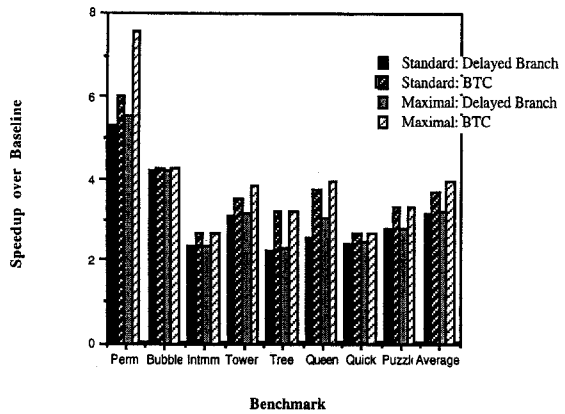

Fig.3 Delayed Branch versus BTC with Scheduled Code

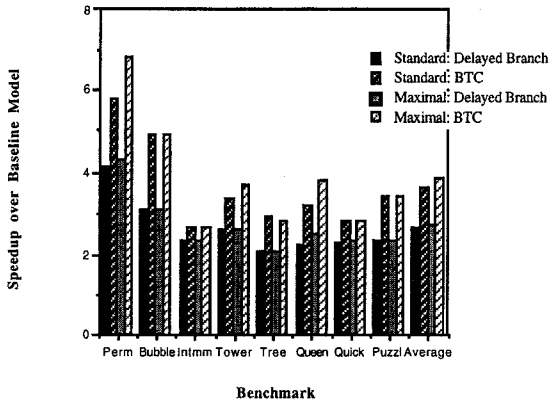

Fig.4 Percentage of Branches Removed in Scheduled Code