# Investigating the Relative Performance of Static and Dynamic Instruction Scheduling

Daniel Tate[1], Gordon Steven[1], Paul Findlay[2]

[1] Information Sciences, University of Hertfordshire
College Lane, Hatfield, Hertfordshire AL10 9AB, England
{D.1.Tate@herts.ac.uk}
{G.B.Steven@herts.ac.uk}
[2] Science and Technology Research Centre, University of Hertfordshire
College Lane, Hatfield, Hertfordshire AL10 9AB, England
{P.A.Findlay@herts.ac.uk}

*Abstract*—

**There are two distinct groups of research into ILP. Those that strongly favour static instruction scheduling and those that favour dynamic instruction scheduling. This paper introduces powerful static and dynamic scheduling models and combines them within the framework of a single simulation environment.**

**Both individual models achieve respectable speedups; dynamic scheduling significantly out-performs static scheduling when an idealised processor model with perfect branch prediction is used. However, when a realistic branch predictor is substituted, the roles are reversed, and static scheduling achieves the higher performance. Similarly, static scheduling performs better in the absence of branch prediction or when processor resources are restricted.**

**Finally, we combine static scheduling with out-of-order instruction issue. Disappointingly, when an ideal out-of-order processor is used, scheduled code fails to match the performance of unscheduled code. Furthermore, with realistic branch prediction, out-of-order issue fails to improve the performance of scheduled code.**

*Keywords*— **High Performance Processors, Instruction Scheduling, Dynamic Scheduling, Multiple Instruction Issue.**

## I. INTRODUCTION

Multiple Instruction Issue can be achieved in high-performance processors through either static or dynamic instruction scheduling. Static scheduling traces its roots to the early 80's [FIS 81] and seeks to exploit a global view of the code while offering simple instruction issue logic and a minimal clock period. Dynamic scheduling originated in the late 60's [TOM 67] and offers massive run-time flexibility at the cost of additional hardware complexity.

The exponents of static instruction scheduling have long insisted that their technology avoids any requirement for out-of-order instruction issue. Instruction schedulers therefore tend to target VLIW processors or, more recently, in-order superscalar processors. Supporters of dynamic scheduling dismiss VLIW as overly restrictive and unworkable. However, with out-of-order instruction issue beginning to reach its technological limits, there is an obvious need for something more. Therefore, while exponents of static instruction scheduling tend to resist any encroachment by dynamic scheduling, the reverse is not true. Instruction scheduling is playing an increasing role in many commercial compilers. This study quantifies the benefits of static and dynamic scheduling within the framework of a single simulation environment.

The first set of results compares the performance of a sixteen-issue out-of-order processor with a sixteen-issue in-order processor. Unscheduled code is tested as well as code that has been dramatically reordered by an aggressive static instruction scheduler. Particular attention is paid to the crucial role of branch prediction; the full spectrum of branch capabilities is examined by simulating no branch prediction, a Branch Target Cache and perfect branch prediction. Finally, the benefits of combining statically scheduled code with out-of-order issue are quantified.

The second set of results examines the effect of reducing the number of functional units available to each processor model. Comparisons are made between the different issue models and the use of static scheduling is evaluated.

This study is based on the Hatfield Superscalar Architecture [STG 97]; an aggressive instruction level scheduler [STF 98] performs static scheduling, while the instruction level simulator [TAT 99] performs dynamic scheduling. All the test results are directly comparable since they are generated by the same simulator [COL 93] through the use of different instruction issue rules and retirement methods.

## II. PREVIOUS WORK

There have been several previous comparisons of static and dynamic scheduling [LOV 90] [MEL 91] [CHA 91a] [LEN 94] [ADV 97]. However, many of these comparisons are handicapped by the absence of a powerful static instruction scheduler, or the use of a relatively low instruction issue rate.

Chang's paper is the most relevant since it is based on the powerful IMPACT compiler [CHA 91b]. Chang compares three main models: restricted static scheduling, full static scheduling and restricted static scheduling with out-of-order issue. The results show comparable performance from the latter two models, with the restricted static scheduling model giving significantly less speedup.

## III. HATFIELD SUPERSCALAR ARCHITECTURE (HSA)

The long term objective of the HSA project is to achieve an order of magnitude speedup over a traditional RISC processor, while minimising the associated increase in code size.

The first incarnation of a wide-issue processor architecture to achieve this aim was the HARP [STG 92]. HARP is a four instruction wide VLIW processor that was designed and fabricated at the University of Hertfordshire. However, as with most VLIW processors, the HARP processor suffers from excessive code expansion and a strict issue model that results in cross-family incompatibilities.

The Hatfield Superscalar Architecture [STG 97] was therefore developed to maintain the promising results achieved by HARP, but also to provide a more flexible model to facilitate further work. The problems of code expansion and the fixed issue rate suffered by HARP were avoided by using a variable issue rate, therefore making the HSA a superscalar architecture. However, the full complexities inherent in out-of-order superscalar architectures were initially avoided by maintaining a strict in-order issue model. The HSA was therefore termed a minimal superscalar architecture. Despite the subsequent addition of both out-of-order instruction issue and dynamic branch prediction capabilities, the underlying in-order model has not been altered.

The HSA supports a basic four-stage in-order pipeline, or a five stage out-of-order pipeline:

| | |
|---|---|
| IF | Instruction Fetch |
| ID/RF | Instruction Decode / Register Fetch |
| EX | Execute |
| WB | Write Result |
| RET | RETire instructions in-order |

The first three stages of the pipeline are the same for both in-order and out-of-order instruction issue models. In the first pipeline stage, instructions are fetched from the instruction cache into an Instruction Buffer. In the second stage register operands are requested from either the integer, floating-point or Boolean register files. Alternatively, the operands can be forwarded from other functional units. Decoded instructions are then issued from the Instruction Buffer to registers ahead of the functional units. In the case of out-of-order instruction issue these registers become Reservation Stations and instructions also have to reserve locations in the Reorder Buffer. In the third stage, instructions pass through the functional units, this stage may require multiple cycles. In the fourth stage, results are written to either the destination registers or the Reorder Buffer, depending on the issue mode. In either case, results are also forwarded directly to other functional units as required. In the fifth stage, instructions that have been issued out-of-order finally return their results to the destination registers. This retirement takes place in order, after instructions have received their results and reach the head of the Reorder Buffer.

To help in the removal of branches during scheduling, the HSA supports guarded, or predicated, instruction execution. The maximum number of guards that may be assigned to each instruction is determined by a constant in the scheduler configuration file; throughout this study the maximum number of guards is two.

To generate code that is executable by the HSA, a Gcc compiler was targeted for the HSA instruction set. The HSA code may then be passed through the Hatfield Superscalar Scheduler [STF 98]. Scheduled or unscheduled code is then executed by a highly parameterised simulator [COL 93]. An equally flexible cache simulator and out-of-order instruction issue capability [TAT 99] have been integrated into the original processor simulator.

## IV. HATFIELD SUPERSCALAR SCHEDULER (HSS)

The main force behind the performance achievements of the in-order instruction issue HSA model is the static instruction scheduler [STF 98]. The HSS receives the HSA instructions as a single sequential stream. It then reorders the instruction stream into parallel instruction groups, where each instruction in a group can be issued and executed in parallel. The instructions are then output as a new sequential stream for processing by the HSA simulator. Assuming an ideal schedule and fetch unit, the simulator's ID stage will then reform the groups identified by the scheduler and issue them in parallel to the functional units for processing.

The HSS achieves high performance by combining an aggressive software pipelining algorithm with a powerful set of low-level code motion primitives. The algorithm can be applied to loops of arbitrary complexity. At the same time, code expansion is controlled in two ways. Firstly, instructions are only moved around loop back edges when it can be shown that the code motion being attempted has the potential to reduce the execution time of the loop. Secondly, loop scheduling is terminated as soon as a minimum loop execution time is achieved.

The scheduling process is divided into five main stages. The first and last stages simply load the unscheduled benchmark into the scheduler's internal data structures and output the scheduled benchmark as a sequential stream of

instructions. The three internal scheduling stages require slightly more attention.

The second stage collects the information needed by the subsequent scheduling stages. This stage creates the initial basic blocks, finds branch targets, detects procedures, detects loops, and computes register live ranges.

The third stage performs optional function in-lining. Functions that satisfy the scheduler's in-lining heuristics are copied into the benchmark at the function call site. Redundant function entry and exit instructions can then be removed.

The software pipelining algorithm is implemented in the fourth stage. The code is scheduled one procedure at a time. In each procedure, innermost loops are scheduled first. This is because programs spend a high proportion of their run-time inside loops. Outer loops are scheduled next, followed by the straight-line code.

The HSS is highly parameterised and able to generate many different types of schedule, as are all of the software projects associated with the HSA. A global configuration file contains all the available switches and parameters. These fall into two main categories: definition of the target processor and specification of the type of schedule required. An example of the former is the number of arithmetic units in a processor. An example of the latter is whether or not to allow code to percolate into a basic block ending with a BSR; the switch 'PERCOLATE_INTO_BSR' controls this decision. Full details are available in [STF 98].

In this study, branch prediction is substituted for the HSA delayed branch mechanism. This change avoids the complications of combining a variable branch delay region with branch misprediction recovery. Scheduling was therefore performed with a branch delay region of zero and the branch instructions themselves were forced to the end of their parallel groups; the scheduling switch FORCE_BRANCH_TO_END was therefore asserted.

## V. SUPERSCALAR SIMULATION ENVIRONMENT

The HSA instruction level simulator is a highly configurable environment. It incorporates facilities for creating, configuring and storing processor models. All facets of the processor may be amended including the cache hierarchy, fetch unit, branch prediction, branch misprediction recovery method, issue policy, register renaming, number and configuration of Reservation Stations, number and piping of functional units, result busses, and retirement method.

The simulator models all processor structures and passes each instruction through all the stages it would pass through in a physical implementation. At each stage of instruction execution, detailed simulation statistics are gathered. At the end of each program run, the statistics are stored in a '.use' file.

Using the in-order instruction issue model an instruction can arbitrate for a functional unit when all of its operands and guards are available. In contrast, using the out-of-order instruction issue model, an instruction can arbitrate for a Reservation Station ahead of a function unit before its source operands are available. A Reorder Buffer is provided to allow the out-of-order instruction issue model to recover from mispredicted branches.

Guarded instructions introduce a further complication. Ideally, instructions should be issued to Reservation Stations irrespective of whether their guards are available. However, when a source operand is not available, Tomasulo's algorithm expects the instruction issue logic to furnish a tag that indicates which instruction will ultimately generate the required operand. If guarded instructions are used, the value of this tag may become indeterminate. For example, consider the following instruction sequence:

```
TB2   ADD  R1, R3, R4
      SUB  R6, R1, R12
```

The ADD will only deliver R1 to the SUB if its Boolean guard evaluates to TRUE at run-time. However, if the guard evaluates to FALSE, an earlier instruction must now deliver the required value. In this paper, the simulator avoids this difficulty by only issuing an instruction to a Reservation Station if all its guards have been resolved. This restriction allows the instruction issue logic to generate unique tags for all unavailable source operands.

The simulator supports three branch prediction models: predict not taken, a conventional Branch Target Cache (BTC), and perfect branch prediction.

Predict not taken does not require any branch prediction logic. Instructions are simply fetched sequentially until a branch is taken. Instructions after the taken branch are then squashed and instruction fetching is restarted at the branch target.

The BTC is direct mapped and has a misprediction rate between 15% and 20%, partly because a return address stack [KAE 91] is not provided to predict subroutine return addresses. The BTC option therefore provides a one-level branch predictor with modest prediction capabilities.

Perfect branch prediction is achieved by generating a program trace for each benchmark. These traces are then used to drive the instruction fetch mechanism when perfect branch prediction is modelled.

Branch mispredictions are handled differently by the in-order and the out-of-order instruction issue models. The in-order instruction issue model has its origins in VLIW and therefore avoids the complexity of out-of-order instruction issue. This enables a branch to complete in the ID or second pipeline stage [STF 93]. Branch mispredictions in an in-order processor can therefore be repaired with only a single cycle delay.

In contrast, the out-of-order model sends branch instructions to Reservation Stations ahead of the branch execution units in the ID stage. The branch is processed in the execution stage. A branch misprediction will therefore incur a two cycle penalty before the new target instructions enter the Instruction Buffer.

Both models resolve mispredicted branches at the earliest opportunity, in the ID stage in the case of in-order instruction issue and in the EX stage with out-of-order instruction issue. A more conservative alternative in the case of out-of-order instruction issue would have been to wait until a branch reaches the head of the Reorder Buffer.

## VI. Test Models

Results are calculated as a speedup factor over a basic RISC processor. The RISC processor can fetch and issue a single instruction in each cycle. It performs no branch prediction, contains one functional unit of each type, and can retire one instruction on any given cycle. On average, the RISC model retires an instruction in 90% of its cycles.

All results presented in this study are given as an average across the eight integer programs from the Stanford benchmark suite. These benchmarks are: Perm, Tower, Sort, Bubble, Queens, Matrix, Tree and Puzzle. Dynamic instruction counts vary between 200 000 and 25 000 000 instructions.

Perfect caches with single cycle access times are assumed throughout this study. All simple integer instructions have a latency of one cycle, the exceptions being multiply which takes three cycles and divide which takes sixteen cycles. There are sixteen integer result busses that pass results to the register files or to the Reorder Buffer, depending on the issue model.

When out-of-order instruction issue is used, a Reorder Buffer is provided with sixty-four entries.

Three branch prediction mechanisms are examined. Firstly, predict not taken continually fetches instructions sequentially until a branch misprediction is signalled. Secondly, a 1K entry Branch Target Cache (BTC) predicts whether each fetch contains a taken branch, and alters the PC accordingly. The size of the BTC ensures that the number of conflict misses is negligible. Finally, perfect branch prediction is achieved by using an instruction trace to drive the simulator's instruction fetch mechanism.

All tests in Section VII are performed using the simulator's 'maximal' model; however, in Section VIII, results are presented for processor models with restricted numbers of functional units. Three additional models are simulated, and will be represented by the letters A-C. The number of resources allocated to each model is shown in Table I. When each model is in use, both the scheduler and the simulator are configured for the number of functional units given in the table. All other parameters are unaffected.

TABLE I
Resource Allocation for Test Models

| Model | Fetch Width | ALU | Shift | Mult | Relat-ional | Load | Store | Branch |
|---|---|---|---|---|---|---|---|---|
| RISC | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| Max | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 |
| A | 16 | 10 | 4 | 2 | 4 | 6 | 4 | 4 |
| B | 16 | 6 | 2 | 1 | 2 | 4 | 2 | 2 |
| C | 16 | 4 | 1 | 1 | 1 | 2 | 1 | 1 |

As with all previous models, there is only one Reservation Station assigned to each functional unit. For example, in model C there are 11 Reservation Stations.

## VII. Initial Results

First we evaluate the impact of static instruction scheduling in the following sixteen-issue processor models:

- In-order with no branch prediction (predict not taken)
- In-order with perfect branch prediction
- Out-of-order with perfect branch prediction

The results are summarised in Fig. 1.

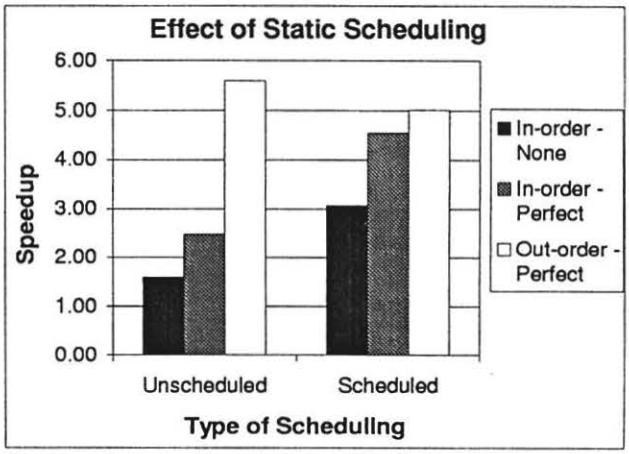


Fig. 1 - Comparison of Unscheduled and Scheduled Benchmarks

The basic MII architecture with no scheduling, in-order instruction issue and no branch prediction achieves a modest average speedup of 1.6. This figure rises to 2.5 when perfect branch prediction is added. However, both these figures are completely eclipsed when out-of-order instruction issue is also added, and an average speedup of 5.6 is achieved.

Scheduled code in turn significantly outperforms unscheduled code as long as in-order instruction issue is used; a 92% improvement in speedup is recorded with no branch prediction and 84% with perfect branch prediction. Nonetheless scheduled code with in-order execution only achieves a speedup of 4.6, and is therefore unable to match

the performance of unscheduled code with out-of-order instruction issue.

Furthermore, with the added benefit of out-of-order instruction issue, scheduled code only achieves a speedup of 5.0, still 11% slower than unscheduled code. However, this is not surprising. The HSS has always targeted an in-order processor model and is therefore unable to take advantage of out-of-order instruction issue. As a result, the HSS introduces additional dependencies as a side effect of the scheduling algorithm, the only check is that the in-order program execution time is not increased. Both register renaming and the addition of Boolean guards introduce further data dependencies that may restrict out-of-order instruction issue. Code expansion is also a factor, although the HSS restricts code expansion to a modest 50%, the out-of-order model still has to issue 50% more code. Finally, Tomasulo's model can not be easily optimised to support guarded execution. As noted earlier guarded instructions are only issued to Reservation Stations after their Boolean guards have been resolved. In an ideal environment with perfect branch prediction, the HSS is unable to match the performance of an out-of-order superscalar running unscheduled code.

Since the role of the branch predictor is clearly crucial, further comparisons are made in Fig. 2 using no branch prediction, a BTC, and perfect branch prediction.

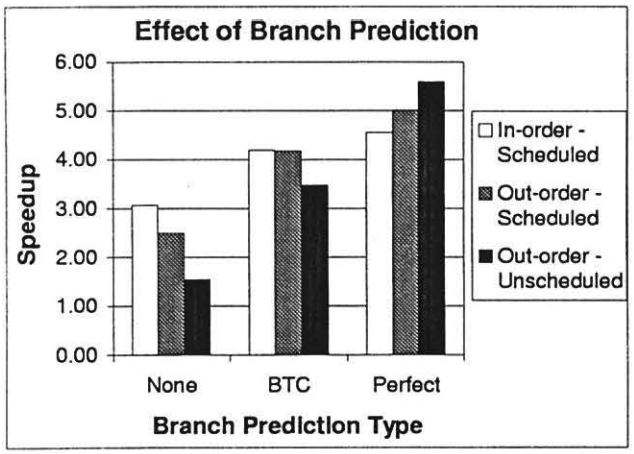


Fig. 2 - Increasing the Performance of the Branch Predictor

In the absence of branch prediction, the instruction scheduler with in-order instruction issue delivers a respectable speedup of 3.1, outperforming the unscheduled out-of-order instruction issue model by a factor of two.

With perfect branch prediction the roles are reversed with out-of-order instruction issue outperforming instruction scheduling by 22%. However, if a one-level BTC is introduced, the instruction scheduler is once more the clear winner achieving a speedup of 4.2 compared to the out-of-order speedup of 3.5. The addition of a BTC has degraded the performance of both scheduled and unscheduled code. However, while performance of the out-of-order unscheduled code has been degraded by 38%, the in-order scheduled code has been degraded by a mere 8%.

An out-of-order processor relies on accurate branch prediction to allow it to assemble parallel instruction groups across basic block boundaries. In contrast, the HSS assembles its parallel groups at compile time and is therefore penalised far less by branch mispredictions. Furthermore, the in-order model has two additional advantages. Firstly, the simpler in-order instruction issue model allows the branch penalty to be reduced, in this study by one cycle. Secondly, about 33% of dynamic branches are removed by the HSS. With fewer branches to predict there are fewer mispredictions; however, this effect is partially offset by an increase in the misprediction rate from 16% to 21%.

These results confirm that the HSS is unable to take advantage of out-of-order instruction issue. Using a BTC, executing scheduled code on an out-of-order instruction issue model fails to deliver any additional speedup.

The impact of branch prediction on each individual processor model is illustrated more clearly in Fig. 3.

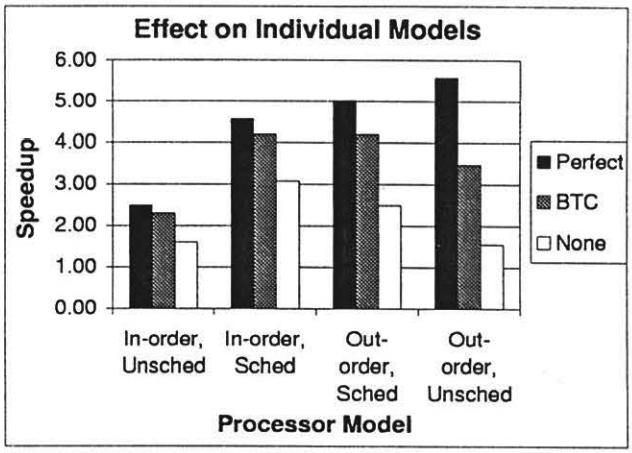


Fig. 3 - Performance of Processor Models

The in-order model is far less effected by decreasing the branch prediction accuracy than the out-of-order model. The in-order model executing either unscheduled or scheduled code is only degraded by 34% when perfect branch prediction is replaced with predict not taken. The out-of-order model is degraded by 73% and 50% when executing unscheduled and scheduled code respectively.

## VIII. Reducing Processor Resources

In this section, we quantify the impact of reducing the number of functional units available to each processor model to more realistic levels. Again three types of branch prediction are simulated: no branch prediction, a BTC and perfect branch prediction. Certain models are significantly more affected by resource limitations than others. To

emphasise these differences, we record how much the execution time of each model is degraded by successive resource restrictions. In all cases the reference point is the overall execution time achieved using the maximal model.

Model A reduces the number of functional units from 112 to 34, see Fig. 4. Since only one Reservation Station is allocated to each functional unit, the number of Reservation Stations is also reduced to 34. In spite of this dramatic reduction, the performance degradation never exceeds 8.5%. The least affected is the in-order model with no branch prediction; the most affected is the out-of-order model with perfect branch prediction.

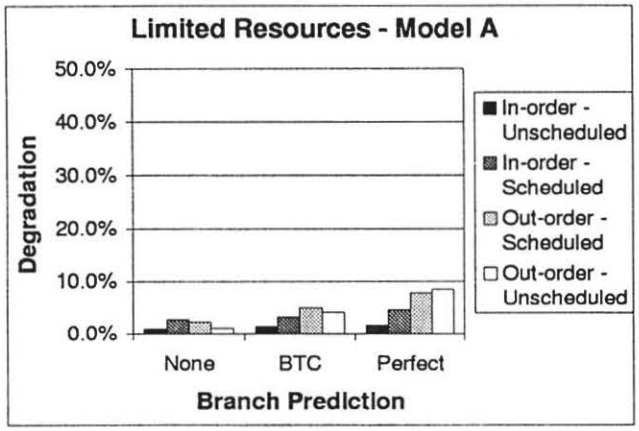


Fig. 4 - Effect of Limiting Resources to Model A

Model B further reduces the number of functional units to 19 and leads to additional performance losses, see Fig. 5.

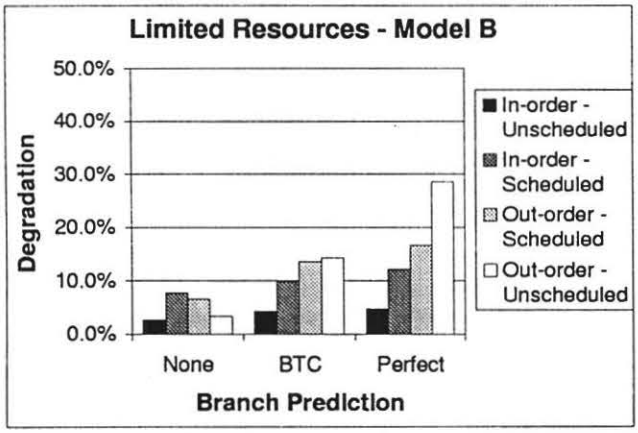


Fig. 5 - Effect of Limiting Resources to Model B

Significantly, the out-of-order unscheduled code with perfect branch prediction is now degraded by 29%. Furthermore, this figure is well over twice the performance loss suffered by in-order scheduled code with perfect branch prediction.

Finally, Model C reduces the number of functional units from 19 to 11. While there are still four arithmetic units and two load units, there is now only one instance of each of the remaining four functional unit types, see Table I. With Model C the performance loss is quite dramatic in many cases, see Fig. 6. Once again the out-of-order unscheduled code is particularly badly hit, loosing 50% of the performance with perfect branch prediction and 35% with a BTC. In contrast, the in-order scheduled code is only degraded by 25% with perfect branch prediction, 22% with a BTC.

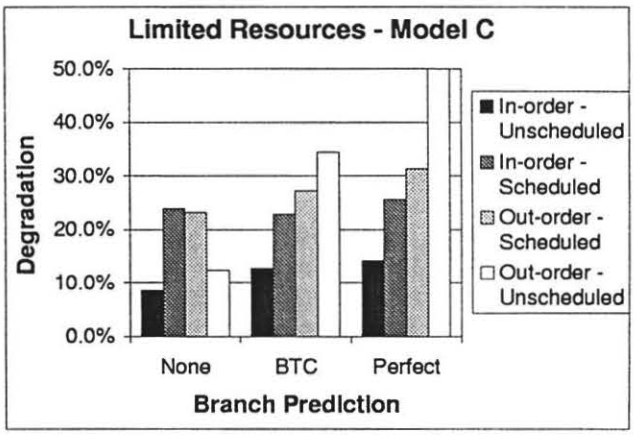


Fig. 6 - Effect of Limiting Resources to Model C

The significance of this massive loss of performance on the speedup of a processor with limited resources is shown in Fig. 7.

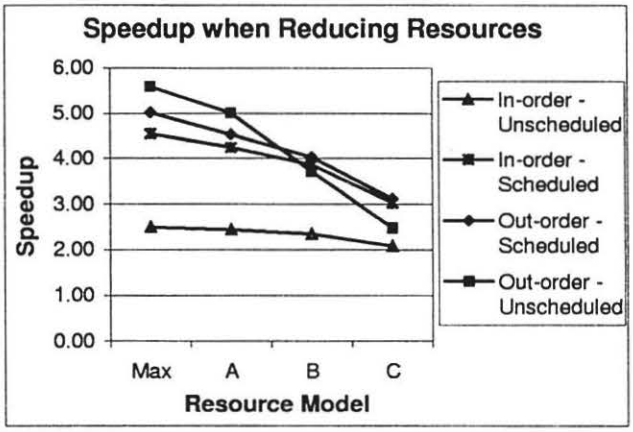


Fig. 7 - Effect of Resources using Perfect Branch Prediction

With the maximal model, executing unscheduled benchmarks out-of-order achieves the highest speedup. Executing scheduled benchmarks out-of-order achieves the second best speedup. The in-order instruction issue model executing scheduled benchmarks comes third. However, with the most restrictive model, the relationships between the speedups of these three models are significantly altered. The out-of-order instruction issue model executing unscheduled benchmarks has slumped from the best

speedup to the worst of the three, and is almost as bad as the in-order instruction issue model executing unscheduled benchmarks. The remaining two models, both of which execute scheduled benchmarks, converge; although the out-of-order instruction issue model maintains a slight speedup advantage of 3.1 to 3.0 when executing on model C.

## IX. Conclusions and Discussion

Until now most of the research in the area of static instruction scheduling has involved VLIW or in-order superscalar architectures. At the same time other research groups have concentrated on refining the hardware of out-of-order superscalar architectures. This compartmentalisation of high-performance processor research into separate hardware and software threads has been quite striking and has a number of important implications. Firstly, there have been surprisingly few comprehensive performance comparisons between statically scheduled processors and out-of-order superscalar processors. Secondly, there has been a remarkable lack of research into instruction scheduling for out-of-order superscalars. Finally, there has been a lamentable dearth of cross-fertilisation between the two major research threads. This paper begins to bridge the two separate research threads by exploring the performance of both statically and dynamically scheduled processor models within the context of a single simulation environment.

The highest average speedup of 5.6 was achieved using a sixteen-issue out-of-order processor model. In contrast, the HSS, our static instruction scheduler, could only achieve an average speedup of 4.6 when targeting a sixteen-issue processor with in-order instruction issue.

However, the above speedups were achieved with perfect branch prediction and no significant resource limitations. When a BTC was substituted for the ideal branch predictor, the better performance was delivered by the HSS. Similarly, when processor resources were restricted, the out-of-order superscalar model was unable to maintain its performance advantage. Our study therefore suggests that out-of-order superscalars are far more sensitive to branch mispredictions and resource restrictions than a statically scheduled processor.

Of course both the out-of-order superscalar model and the HSS can be improved. For example the number of branch mispredictions could be reduced by adding a stack to hold subroutine return addresses and by adopting Two-Level-Adaptive Branch Prediction [YEH 92]. While both models would benefit, these changes are likely to favour the out-of-order model. Also perfect caches were assumed throughout this study. A more realistic cache would also favour the out-of-order processor models, since they are likely to tolerate cache misses better than the HSS.

At the same time the HSS is undergoing continuous improvements. In particular, this study revealed that the HSS is optimised for a processor with branch delay slots rather than the processors with branch prediction used here. In this study the HSS therefore appears to be over-scheduling, a defect that will be rectified in future versions.

It must also be remembered that the additional complexity of the out-of-order model is likely to result in longer processor cycle times and multiple ID pipeline stages. Even a 10% increase in processor cycle time would cancel out 50% of the out-of-order model's advantage in the most favourable case. Similarly, an extra ID stage would increase the branch misprediction penalty and would further emphasise the out-of-order processors inability to tolerate branch mispredictions.

The final striking result in this study was the inability of the HSS to benefit from out-of-order instruction execution. There are several reasons for this. Firstly, Tomasulo's algorithm does not cope well with guarded execution. This is clearly an area requiring further research. Secondly, the HSS introduces additional data dependencies as a side effect of the scheduling process. These must be reduced if an out-of-order processor is to be successfully targeted. Finally, although the HSS removes branches as a side effect of the scheduling process, an out-of-order issue processor would benefit from more aggressive branch removal.

## X. References


[ADV 97] ADVE S V, et al. *Changing Interaction of Compiler and Architecture.* Computer Magazine, Vol.30 No.12, December 1997. pp 51-58.

[CHA 91a] CHANG P P, CHEN W, MAHLKE S, HWU W. *Comparing Static and Dynamic Code Scheduling for Multiple-Instruction-Issue Processors.* Micro-24, Albuquerque, New Mexico, November 1991. pp 25-33.

[CHA 91b] CHANG P P, MAHLKE S, CHEN W, WARTER N J. HWU W. *IMPACT: An Architectural Framework for Multiple-Instruction-Issue Processors.* 18th Annual International Symposium on Computer Architecture. Toronto, May 1991. pp 266-275.

[COL 93] COLLINS R. *Developing a Simulator for the Hatfield Superscalar Processor.* University of Hertfordshire Technical Report No.172, December 1993.

[FIS 81] FISHER J A. *Trace Scheduling: A technique for global microcode compaction.* IEEE Transactions on Computers, Vol.C-30 No.7, July 1981. pp 37-47.

[KAE 91] KAELI D R, EMMA P G. *Branch History Table Prediction of Moving Target Branches due to Subroutine Returns.* 18th Annual International Symposium on Computer Architecture, Toronto, May 1991. pp 34-41.

[LEN 94] LENELL J, BAGHERZADEH N. *A Performance Comparison of Several Superscalar Processor Models with a VLIW Processor.* Microprocessors and Microsystems, Vol.18 No.3, April 1994. pp 131-139.

[LOV 90] LOVE C E, JORDAN H F. *An Investigation of Static Versus Dynamic Scheduling.* 17th Annual

International Symposium on Computer Architecture, Seattle, Washington, June 1990. pp 192-201.

[MEL 91] MELVIN S, PATT Y N. *Exploiting Fine-Grained Parallelism Through a Combination of Hardware and Software Techniques.* 18th Annual International Symposium on Computer Architecture, Toronto, Canada, May 1991. pp 287-296.

[POT 98] POTTER R D. *Exploring the Limitations of the Fine Grained Parallelism in a Superscalar Architecture.* PhD Thesis, University of Hertfordshire, 1999.

[STF 93] STEVEN F L, ADAMS R G, STEVEN G B, WANG L, WHALE D J. *Addressing Mechanisms for VLIW and Superscalar Processors.* Microprocessing and Microprogramming, Vol.39 Numbers 2-5, December 1993. pp 75-78.

[STF 98] STEVEN F L. *An Introduction to the Hatfield Superscalar Scheduler.* University of Hertfordshire Technical Report No.316, Spring 1998.

[STG 92] STEVEN G B, ADAMS R G, FINDLAY P A, TRAINIS S A. *iHARP: A Multiple Instruction Issue Processor.* IEE Proceedings, Part E, Computers and Digital Techniques, Vol.139 No.5, September 1992. pp 439-449.

[STG 97] STEVEN G B, CHRISTIANSON D B, COLLINS R, POTTER R D, STEVEN F L. *A Superscalar Architecture to Exploit Instruction-Level Parallelism.* Microprocessors and Microsystems, Vol.20 No.7, March 1997. pp 391-400.

[TAT 99] TATE D. *Out-of-Order Instruction Issue and its Integration into the Hatfield Superscalar Architecture.* University of Hertfordshire Technical Report No.330, April 1999.

[TOM 67] TOMASULO R M. *An Efficient Algorithm for Exploiting Multiple Arithmetic Units.* IBM Journal of Research and Development, January 1967. pp 25-33.

[YEH 92] YEH T, 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.