DIVISION OF COMPUTER SCIENCE

An Evaluation of the iHARP Multiple Instruction Issue Processor

Fleur L Steven
Gordon B Steven
Liang Wang

Technical Report No.179

February 1994

An Evaluation of the iHARP Multiple Instruction Issue Processor

Fleur L Steven, Gordon B Steven and Liang Wang
Division of Computer Science
University of Hertfordshire

ABSTRACT

RISC processors have approached an execution rate of one instruction per cycle by using pipelining to speed up execution. However, to achieve an execution rate of more than one instruction per cycle, processors must issue multiple instructions in each processor cycle. This paper evaluates the architectural features of iHARP, a VLIW (Very Long Instruction Word) processor with an instruction issue rate of four, which has been developed at the University of Hertfordshire. One of the distinctive features of iHARP is the provision of Boolean guards on all instructions. Every iHARP instruction is only executed at run time if the attached Boolean guard is true. This paper evaluates the benefits of guarded instruction execution and quantifies its performance advantage. Other architectural features considered include instruction issue rate, code size, number of data cache ports, number of register file write ports, number of branch units and addressing mechanisms. The evaluation uses RLS, a resource limited instruction scheduler, specifically developed to statically reorder code for parallel execution on iHARP.

Key Words: VLIW Superscalar Instruction Scheduling Guarded Instruction Execution

1. INTRODUCTION

iHARP is a multiple instruction issue (MII) processor which fetches a 128 bit long instruction word from an instruction cache in each cycle. Each long instruction defines four, 32-bit RISC primitives which are dispatched to four integer pipelines for parallel execution. iHARP is therefore a VLIW processor which relies on a software instruction scheduler to detect groups of instructions which can be executed in parallel and to place these groups into a long instruction word at compile time. This approach contrasts with the superscalar [1] approach where it is left to the hardware to detect instructions which can be executed in parallel at run time.

This paper describes the development of RLS, a resource limited instruction scheduling system, and its use in the evaluation of the iHARP architecture [2]. The features considered include instruction issue rate, code size, number of data cache ports, number of branch units, number of register file write ports and addressing mechanisms.

The usefulness of the HARP conditional execution facility, which allows a Boolean guard to be attached to every iHARP instruction, is also quantified. Although guarded instruction execution has been widely proposed in the literature, quantitative evaluations of the benefits of guarded execution are not widely available.

The remainder of this paper is organised as follows: Section 2 discusses the impact of data dependencies on instruction scheduling; section 3 reviews other research in the area of static instruction scheduling; section 4 describes the compiler/scheduler software and the HARP models used in this research; section 5 contains our evaluation of iHARP and presents our results; section 6 offers some concluding remarks.

2. DATA DEPENDENCIES

Scheduling instructions for parallel execution can be viewed as a process in which each instruction is successively moved or percolated [3] up through the code structure in an attempt to ensure that it is executed at the earliest possible opportunity. This code motion is ultimately stopped by data dependencies between pairs of instructions.

Three classes of data dependencies can be identified: Read after write (RAW), write after read (WAR) and write after write (WAW). However, only RAW dependencies represent true data dependencies and therefore ultimately limit the performance of MII processors. In contrast, WAR and WAW data dependencies can both be removed by using register renaming.

For example, in the instruction sequence below instruction I2 has a WAR or anti-dependence on I1.

```
I1      ADD R5, R6, R7
I2      ADD R6, R8, #256
```

This dependence can be removed by returning the result of I2 to an unallocated register, in this case R20. This renaming allows I2 to be percolated ahead of I1 in the instruction schedule:

```
I2      ADD R20, R8, #256
I1      ADD R5, R6, R7
        ..
        MOV R6, R20
```

The move instruction is required to restore the new result to R6. Note this extra instruction need not introduce further data dependencies since subsequent instructions using R6 can equally well use R20.

Register renaming can also be used to remove spurious data dependencies which arise when code is moved between basic blocks. Consider the following example:

```
        NE B6, R1, R2 ; set B6 if R1 <> R2
        BT B6, Label  ; if B6 is true goto Label

Label:
        LD R6, 8(SP)
```

The LD instruction could be moved ahead of the branch instruction and executed speculatively[1] giving the following code:

```
        NE B6, R1, R2
        LD R6, 8(SP)
        BT B6, Label

Label:
```

Unfortunately if R6 is live on the alternative path,

[1] An instruction is executed speculatively if it is executed before it is known whether the path containing the instruction will actually be taken.

it will have been incorrectly updated whenever the branch fails. Register renaming can be used to avoid this problem:

```
        NE B6, R1, R6
        LD R20, 8(SP) ; R6 replaced by R20
        BT B6, Label

Label:
        MOV R6, R20
```

As before, a MOV instruction is required to copy the contents of R20 into R6 if the branch is taken.

An alternative solution is to use guarded instruction execution. On iHARP any of the eight Boolean registers which are used to record the results of a relational instruction can also be used as Boolean guards. In the above example B6 can therefore be used to guard the execution of the LD instruction:

```
        NE B6, R1, R2
TB6     LD R6, 8(SP)    ; Execute load if B6 is true
        BT B6, Label

Label:
```

Now the Boolean guard ensures that the LD will only be executed if the branch is taken.

Any further code motion will move the LD instruction beyond the scope of the Boolean guard. Now only register renaming can be used to remove any data dependence:

```
        LD R20, 8(SP) ; R6 replaced by R20
        NE B6, R1, R6
        BT B6, Label

Label:
        MOV R6, R20
```

The above code illustrates a further problem introduced by the speculative execution of instructions. Suppose the load instruction in the previous example generates an invalid memory reference. If the path originally containing the load instruction is not actually followed, the instruction will generate a spurious exception which will incorrectly terminate the program.

To solve this problem, all non-branch instructions must exist in two forms. In the first form, any exception generated by an instruction is immediately taken in the usual way. In the second speculative form, an exception will cause a polluted value to be loaded into the instruction's result register. For example, consider the code below:

```
        BT B6, Label
        LD R6, 8(SP)
        SUB R8, R6, #1
        NE B3, R8, #0

Label:
```

Now assume that both the load and subtract instructions are scheduled speculatively ahead of the branch instruction:

```
LD! R6, 8(SP)     ; speculative load
SUB! R8, R6, #1   ; speculative subtract
BT B6, Label
NE B3, R8, #0

Label:
```

If the load instruction generates an exception, R6 will be marked as polluted. Since the subtract instruction is also executed speculatively, it will in turn mark R8 as polluted when it attempts to use the polluted value in R6. An exception will only be taken when the non-speculative relational instruction attempts to use the polluted value held in R8. Note this is the earliest point in the code where we can be certain that the speculative load should have been executed.

To support speculative execution an extra bit must be added to all processor registers, including the Boolean registers, to mark polluted values. This hardware support allows loads and other instructions, such as adds which generate an exception on overflow, to be executed speculatively. However, store instructions can still not be executed speculatively. Stores can only be safely percolated into a preceding basic block if they can be guarded.

3. INSTRUCTION SCHEDULING

This section first considers the amount of parallelism that is ultimately available to MII processors and then reviews current work in instruction scheduling.

3.1 Potential Instruction Level Parallelism

Wall [4] used simulations based on instruction

traces to investigate the parallelism available to superscalar processors. Even with perfect renaming and memory disambiguation, the parallelism realised rarely exceeded seven and was typically only five. However, when Wall substituted perfect branch prediction for his hardware branch prediction model, the amount of parallelism realised increased spectacularly.

Yale Patt's [5] group also used trace driven simulations to investigate superscalar performance. The group concluded that current technology could achieve execution rates of between two and six instructions per cycle and looked forward to significantly faster execution rates in the future.

By far the most spectacular upper bounds on fine-grained parallelism were reported by Lam [6]. Lam recognised that superscalar processors, which rely on hardware to exploit parallelism, can only extract parallelism between successive branch mispredictions. Avoiding this restriction significantly increases the available parallelism. Lam's results range from two instructions per cycle for her basic model to 159 instructions per cycle for an ORACLE model with perfect branch prediction.

All of the above studies emphasise that significantly more parallelism is available to MII processors than is realised in current designs. This gap reflects both limited hardware resources and the current early stage of development of instruction scheduling technology.

The role of accurate branch prediction is also emphasised, reflecting the inability of current superscalars to extract parallelism across mis-predicted branches. However, static instruction scheduling can remove this dependence on branch prediction by percolating instructions across branches from both successor basic blocks. Parallelism is then ultimately limited only by true data dependencies, resources, unresolved memory ambiguities and, of course, by the scheduling technology.

3.2 Scheduling Techniques

In general, global instruction scheduling can be divided into low-level and high-level components. Low-level code scheduling is concerned with the movement of individual instructions subject to data dependencies and resource constraints. High-level scheduling involves scheduling precedence, transformations of the program graph and applications of low-level scheduling. Low-level scheduling techniques can be divided into two categories: List Scheduling [7] and the core transformations of Percolation Scheduling [3]. To fully realise the benefits of List Scheduling, the high-level must enlarge the scheduling scope before List Scheduling is applied. Forming traces in Trace Scheduling [8] and Instruction Boosting [9], unrolling loops, inlining functions and enlarging basic blocks can all be viewed as high-level techniques which enlarge the scope for List Scheduling. List Scheduling can also be used to schedule loops and basic blocks in Lam's software pipelining and Hierarchical Reduction [10].

The core transformations, on the other hand, attempt to move individual instructions upwards through the flow graph to seek parallelism between an instruction and its predecessors. Data dependencies and hardware constraints are therefore checked dynamically as each instruction is moved. Enhanced Percolation Scheduling [11] is a high-level scheduling technique which applies the core transformations to cyclic as well as acyclic code. The distinctive features of Enhanced Percolation Scheduling are the concept of tree instructions, code percolation across loop back edges and the equal treatment of branch successors.

4. THE HARP PROJECT

The aim of the HARP project was to develop an MII architecture which could sustain an instruction execution rate significantly in excess of one instruction per cycle. As part of the project iHARP [12], a VLIW processor with an issue rate of four, was designed, fabricated and tested.

4.1 iHARP Processor

iHARP provides 32, general-purpose registers and eight, 1-bit boolean registers. Four parallel pipelines and a memory unit share the general purpose register file.

iHARP uses a four-stage pipeline. In the IF stage four instructions are fetched from the instruction cache; in the RF stage register operands are accessed; in the ALU/MEM stage instructions are executed or access the data cache; finally in the WB stage results are returned to the register file. To allow the data cache to be accessed in the third pipeline stage, all addresses are formed in the RF stage. Complete operand bypassing then allows the results of ALU and load

instructions to be used in the following cycle. Since branch conditions are also resolved in the RF stage, the branch delay is one.

While all pipelines are able to support ALU, relational and a limited range of shift instructions, the functionality of each pipeline is inevitably restricted by the need to conserve hardware resources. As a result only two pipelines, one and three, support branch instructions. Similarly memory reference instructions are restricted to pipelines zero and two. However, although two branches can be executed in parallel, the single data cache port precludes the parallel execution of two memory references instructions. Finally 32-bit literals are introduced by providing 64-bit instructions which occupy two adjacent instruction positions.

4.2 MII HARP Architectures

To evaluate the HARP architecture, a family of MII architectures with different instruction issue rates was postulated. Each model has the same instruction set, addressing modes and pipeline stages as iHARP and provides hardware support for speculative instruction execution. It is assumed that every pipeline will support ALU, relational and shift operations. Also, in line with iHARP, a maximum of two branch instructions and one memory reference instruction can be issued in each cycle. These base parameters were then systematically varied to evaluate the architecture.

4.3 HARP Compiler and Instruction Scheduler

A HARP C compiler was generated using the GNU CC compiler generator [13]. The sequential HARP code produced by the compiler was then scheduled for multiple instruction issue using a Resource Limited Scheduler developed by Liang Wang [2]. The HARP simulator [14] was used to execute both sequential and parallel code.

RLS is a resource limited scheduler which was developed specifically to exploit fine-grained parallelism in iHARP. Since iHARP has only four pipelines, clearly the scheduler has only limited resources at its disposal. Furthermore, VLIW architectures inevitably expand code size since, in practice, it is impossible to fill every long instruction with useful operations. RLS aims to control this code expansion by ensuring that the number of long instructions generated never exceeds the original number of sequential instructions. The scheduler is therefore also 'resource limited' in this sense.

Conceptually RLS consists of a high level and a low level. The high level transforms the sequential instructions of a procedure into a linked data structure, detects basic blocks, constructs a flow graph and unrolls simple loops. It then uses a set of heuristics to select the next basic block for the low-level scheduler.

The low-level scheduler percolates individual instructions from a basic block into an instruction graph of previously scheduled instructions. Guarded instruction execution, renaming and memory disambiguation are all used to increase parallelism. Also after each loop has been scheduled, an attempt is made to move code across the loop back edge to reduce the size of the loop body and to achieve software pipelining.

5. iHARP EVALUATION

This section evaluates the HARP architecture using the well-known Stanford set of integer benchmarks. The HARP model with an issue rate of one is used as a reference model and is referred to as the HARP RISC. To ensure a fair comparison, a single pipeline scheduler [2] was first used to fill the branch delay slots in the sequential HARP code. On average filling the delay slots improved the performance of the HARP RISC by 7.3%. All the speedup figures presented are relative to the HARP RISC after, and not before, the branch delay slots have been filled.

5.1 Instruction Issue Rate

Fig 5.1 shows the average speedups achieved by RLS as a function of instruction issue rate. Each model is assumed to have an ALU per pipeline, two branch units and a single data cache port. The average speedups obtained for issue rates of two, three, four, five and eight are 1.45, 1.66, 1.74, 1.76 and 1.77 respectively. RLS therefore performs well for its four-pipeline target processor, but fails to deliver significant additional parallelism as further pipelines are added.

These figures compare favourably with other groups working in the area. For example, the IMPACT group [15] obtained a speedup of 1.6 and 2.00 for issue rates of two and four respectively.

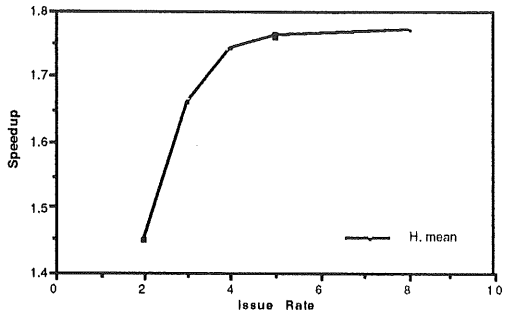

Fig 5.1 Speedup over HARP RISC

Code size increased by 1.38, 1.86, 2.34 and 2.82 for issue rates of two, three four and five (Fig 5.2). RLS therefore comfortably achieves its design target of ensuring that the number of long instruction words after scheduling never exceeds the initial sequential instruction count.

One of the main disadvantages of a VLIW processor is the disproportionate number of NOPs introduced into the code. It is therefore instructive to compare HARP with a minimal superscalar architecture [16] executing scheduled HARP code from which all the NOPs have been removed[2] . The minimal superscalar processors have the same instruction fetch rate and functional capabilities as the equivalent HARP model. However, the superscalars fetch instructions into an instruction window and then issue multiple independent instructions from the window to the functional units. If the superscalar is restricted to in-order instruction issue and if the maximum instruction issue rate equals the fetch rate, the superscalar will, in the worst case, simply reconstruct the long instruction word schedule generated for HARP. Performance will therefore equal or exceed HARP at all issue rates. However, since all the NOPs have been removed, code size will be dramatically reduced to the number of operations in the HARP code. Code expansion would then be only 15% for an issue rate of two, rising to 18% for an issue rate of five (Fig 5.2).

[2] To preserve the semantics of the program, minor reordering of instructions within each LIW would also be required.

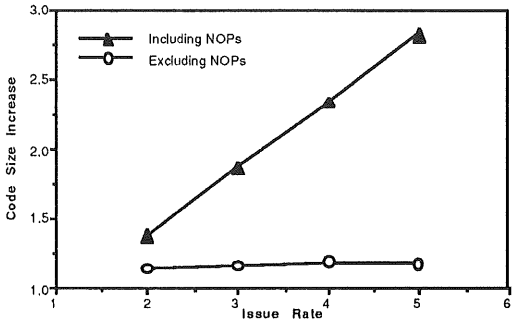

Fig 5.2 Code Size versus Issue Rate

5.2 The number of cache memory ports

Some of the benchmarks execute a high percentage of memory reference instructions. For these programs a single data cache port represents an obvious performance bottleneck, particularly as the instruction issue rate is increased. In spite of the high cost, it is therefore useful to consider adding additional data cache ports. With two data cache ports, the speedups range from 1.51 with an issue rate of two to 1.97 with an issue rate of eight (Fig 5.3), and the average performance of a four pipeline processor is improved by approximately 10%. With three cache ports, the speedups range from 1.84 with an issue rate of three to 2.04 with an issue rate of eight (Fig 5.3), and the average performance of a four pipeline processor is improved by approximately 14%.

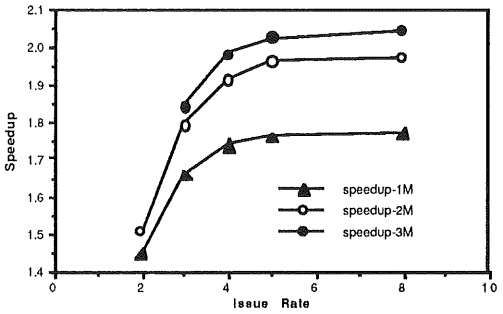

Fig 5.3 Speedup vs number of data cache ports

The improvements recorded varied widely between individual benchmarks. Four 'memory intensive' programs, *perm*, *bubble*, *queens* and *tower*, all achieved significant gains. For example, with an issue rate of four, the addition of a second port improved the execution time of *perm* by 27%. In contrast, two 'ALU intensive' programs *intmm* and *puzzle* obtained virtually no benefit from additional data cache ports.

5.3 Guarded instruction execution

Guarded instruction execution has been proposed by a number of people including Hsu and Davidson [17] and was incorporated in the pioneering Acorn ARM processor [18]. Guarded instructions also form an integral part of the HARP architecture. RLS was therefore designed to make optimum use of both guarded instruction execution and register renaming. However, to evaluate their relative merits, RLS can also rely solely on either guarded execution or renaming.

Fig 5.4 and 5.5 demonstrate the performance benefits of guarded instruction execution. With one memory port, the speedups obtained using both register renaming and guarded execution range from 1.45 to 1.77. These figures fall to 1.45 and 1.68 if only guarded execution is used and to 1.38 and 1.63 if only renaming is used (Fig 5.4).

Similar results are recorded with two data cache ports (Fig 5.5). With both renaming and guarded instruction execution, speedups range from 1.51 to 1.97. These figures then fall to 1.51 and 1.84 with guarded execution only and to 1.44 and 1.72 with renaming only.

Since, in practice, there is little point in using guarded execution on its own, these figures suggest that using guarded execution will improve performance by 9.4% with one memory port and by 14.4% with two memory ports.

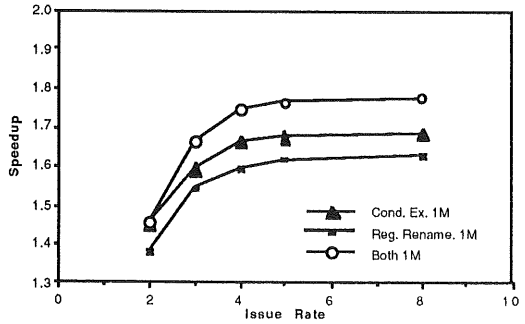

Fig 5.4 Impact of guarded execution using one memory port

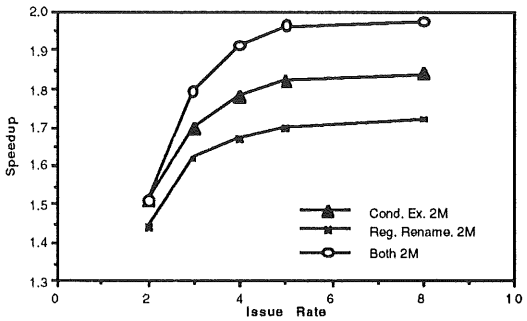

Fig 5.5 Impact of guarded execution using two memory ports

Both renaming and guarded execution have advantages and disadvantages. Renaming has two advantages. First, it is more flexible, since it supports code motion across multiple branch edges; second, it can be used to remove RAW and WAW dependencies. The disadvantage is that it introduces additional restoring code. As well as consuming resources, these copy instructions introduce new data dependencies which the scheduler may then not be able to remove.

Guarded instruction execution has three main advantages: First it avoids restoring code and therefore conserves resources; second it avoids using additional registers; third it allows store instructions to be percolated across conditional branch edges. Sole use of guarded execution also avoids live variable analysis and speculative execution. There are two main disadvantages: First guarded execution introduces a new data dependence between the boolean guard and the instruction being moved; second guarded execution can not be used to remove WAR and WAW data dependencies.

When RLS scheduled the Stanford benchmarks, there were always sufficient registers available to support renaming. Also the flat nature of the speedup curves at the higher issue rates suggests that resources were not a problem. The advantage of guarded execution over renaming has therefore three possible explanations. First, the additional copy instructions added by renaming introduced further data dependencies which the scheduler was unable to remove. Second, guarded instruction execution allowed store instructions to be percolated across branch edges. Finally since RLS was conceived as a scheduler for iHARP, it is possible that RLS is biased towards guarded instruction execution.

5.4 ORed Indexing

A further distinctive feature of iHARP is the use of ORed indexed addressing. In ORed indexing the two memory address components are logically ORed together instead of being added in the conventional manner. This simplified addressing mechanism allows all address computations to be performed in the RF stage and gives rise to the compact four-stage pipeline with no load delay described earlier in this paper. ORed indexing was investigated in a paper presented at Euromicro93 [19] where it was demonstrated that its use typically boosted HARP performance by 10%.

5.5 The number of register writeback ports

Throughout this study it has been assumed that sufficient write ports were always provided on the general-purpose register file to allow all results to be returned to a register in the final pipeline stage. This is equivalent to assuming that the number of write ports is always equal to the number of pipelines.

In practice, multiple write ports can be costly to implement. The performance impact of reducing the number of write ports in a four pipeline model was therefore investigated. On average, reducing the number of write ports from four to three degraded performance by a negligible 0.6%, while reducing the number of ports to two reduced performance by only 4.6%.

The iHARP chip was implemented with only two write-back ports on the register file. This design compromise therefore reduced performance by less than 5%. An additional mechanism was provided on iHARP to enable results to be bypassed directly to the following long instruction word without being written to the register file [12]. The objective was to reduce the pressure on the register file write ports. However, RLS has not attempted to use this facility, since the maximum possible gain is less than 5%.

5.6 Parallel Execution of Branch Instructions

This study has also assumed that two branch units were always available, allowing two branch instructions to execute in parallel. Surprisingly, removing the second branch unit has a negligible impact on performance, less than 0.25% on average. This result may be partially attributed to the ability of RLS to schedule branches in branch delay slots. Although such scheduling is unusual, it is easily achieved in iHARP by adding a Boolean guard to the branch placed in the delay slot.

6. CONCLUSIONS

This paper has described RLS, a resource limited scheduler, which has been used to evaluate iHARP, a VLIW processor with an issue rate of four. With four pipelines a speedup of 1.76 was achieved using non-numeric benchmarks. Significantly, this performance was based on an architecture which has been implemented in silicon, and not on an abstract model. Also the speedup recorded is relative to optimised single pipeline code where branch delay slots have been filled wherever possible.

The above performance improvement was achieved by increasing code size by 134%. Significantly, this increase could be reduced to only 18% by moving to an equivalent minimal superscalar architecture.

A single memory port is an obvious bottleneck in a processor with an issue rate of four. Providing two data ports improved performance by 10% while three data ports improved performance by 14% and achieved a speedup of two.

The benefits of guarded instruction execution were also quantified. Our figures give guarded execution an advantage of 9.4% and 14.4% over renaming with one and two memory ports respectively. While these figures are encouraging, we feel that it is premature to come to any firm conclusion regarding guarded execution. In particular, it will be interesting to see how guarded execution performs with more aggressive scheduling algorithms and higher instruction issue rates.

Other results suggest that our original decision to incorporate ORed indexing in iHARP was fully justified and provided a 10% boost in performance. Similarly, restricting the register file to two write back ports did not significantly degrade performance. Finally, we note that RLS was not able to make significant use of the two parallel branch units in iHARP.

ACKNOWLEDGEMENTS

The authors would like to thank the rest of the HARP team, in particular, Roger Collins, Corrie Elston, Rod Adams and Dave Whale from

Computer Science and Paul Findlay, Brian Johnson and Dave McHale from Electrical Engineering. They would also like to thank Professor M Loomes, Dr S L Stott, J A Davis and Professor P Kaye for their support throughout the HARP project. The HARP project is supported by SERC Research Grant GR/F88018.

REFERENCES

1. Johnson M "Superscalar Microprocessor Design, Prentice Hall, 1991.
2. Wang L "Instruction Scheduling for a Family of Multiple-Instruction-issue Architectures," PhD Thesis, University of Hertfordshire, Dec. 1993.
3. Nicolau A "Uniform Parallelism Exploitation in Ordinary Programs," Proc. of Int. Conf. on Parallel Processing, Aug. 1985, pp614-618.
4. Wall D W "Limits of Instruction-Level Parallelism," ASPLOS IV, April 1991, pp 176-188.
5. Butler M, Yeh Tse-Yu, Patt Y, Alsup M, Scales H and Shebanow M "Single Instruction Stream Parallelism is Greater than Two," 18th Ann. Int. Symp. on Computer Architecture, Toronto, May 1991, pp 276-286.
6. Lam M S & Wilson R P "Limits of Control Flow on Parallelism," 19th Ann. Int. Symp. on Computer Architecture, Gold Coast, Australia, May 1992, pp 46-57.
7. Landskov D, Davidson S, Shriver B & Mallett P W "Local Microcode Compaction Techniques," Computing Surveys, Vol. 12, No. 3, September 1980, pp 261-294.
8. Fisher J A "Trace Scheduling: a technique for global microcode compaction," IEEE Transactions on Computers, C-30, (7), July 1981, pp 478-490.
9. Smith M D, Horowitz M & Lam M "Efficient Superscalar Performance Through Boosting," ASPLOS V, October 1992, pp 248-259.
10. Lam M S "Software pipelining: an effective scheduling technique for VLIW machines," SIGPLAN 88 Conference of Programming Language Design and Implementation, Georgia, USA, June 1988, pp 318-328.
11. Moon S & Ebcioglu K "An Efficient Resource-Constrained Global Scheduling Technique for Superscalar and VLIW Processors," Micro25, Portland, Oregon, December 1992, pp 55-71.
12. Steven G B, Adams R G, Findlay P A & Trainis S A "iHARP: a multiple instruction issue processor," IEE Proceedings-E, Vol. 139, No. 5, September 1992, pp439-449.
13. Stallman R M "Using and Porting GNU CC", Free Software Foundation, 1989.
14. Whale D J "Development of a Processor Simulation for iHARP," Division of Computer Science, University of Hertfordshire, April 1992.
15. Chang P P, Mahlke S A, Chen W Y, Warter N J & Hwu W W "IMPACT: An Architectural Framework for Multiple-Instruction-Issue Processors," 18th Ann. Int. Symp. on Computer Architecture, Toronto, May 1991, pp 266-275.
16. Collins R "A Simulator for the HSP Superscalar Processor", Division of Computer Science Technical Report No. 172, University of Hertfordshire, 1993.
17. Hsu P Y T & Davidson E S "Highly concurrent scalar processing," 13th Ann. Int. Symp. on Computer Architecture, 1986, pp 386-395.
18. Furber S "VLSI RISC Architecture and Organization," Marcel Dekker, 1989.
19. Steven F L, Adams R G, Steven G B, Wang L & Whale D J "Addressing mechanisms for VLIW and superscalar processors," Euromicro 93, Barcelona, September 1993, pp 75-78.