A Parallel Pipelined Processor with Conditional Instruction Execution

Rod Adams and Gordon Steven
Division of Computer Science
Hatfield Polytechnic
College Lane
Hatfield
Herts AL10 9AB
UK

Abstract

In a recent paper by Smith, Lam and Horowitz [1] the concept of 'boosting' was introduced, where instructions from one of the possible instruction streams following a conditional branch were scheduled by the compiler for execution in the basic block containing the branch itself. This paper describes how code from **both** instruction streams following a conditional branch can be considered for execution in the basic block containing the branch. Branch conditions are stored in Boolean registers and all instructions are conditionally executed based on the value in a Boolean register. The two instruction streams can therefore be executed on complementary values of the same Boolean register.

1. Introduction

HARP (the HAtfield Risc Processor) [2,3] is a parallel pipelined RISC processor currently being developed at Hatfield Polytechnic. The major aim of the HARP project is to develop a processor which will execute non-scientific programs at a sustained instruction execution rate in excess of one instruction per cycle.

This goal will be achieved by the simultaneous development of the processor architecture and an optimising compiler. The compiler's role is to schedule independent instructions for a set of parallel pipelines provided by the processor. The compiler detects independent short RISC-type instructions which can be executed in parallel and packs them into long instruction words. The processor fetches these long instruction words from an instruction cache and passes the component short instructions independently through the multiple pipeline structure.

In HARP all instructions are executed conditionally. This conditional execution is used to increase the amount of realisable parallelism in general purpose code. For example, it allows instructions from both instruction streams following a conditional branch to be moved or copied into the basic block containing the branch instruction. Conditional execution is made possible by using Boolean registers to hold the result of a compare instruction. The two instruction streams can then be executed conditionally on complementary values of the same Boolean register.

This paper outlines the major ideas behind the HARP concept. Section two of this paper expounds the HARP approach to the production of a parallel pipelined RISC processor, while section three briefly introduces the major features of the HARP architectural model [4]. The main section, section four, discusses how the compiler makes use of conditional instruction execution to schedule additional instructions in parallel. Finally, in section five, a brief comparison is given between conditional execution and the instruction boosting scheme proposed in the paper by Smith, Lam and Horowitz [1].

2. The HARP Approach

Unlike most other LIW (Long Instruction Word) processors, HARP aims to exploit the low-level parallelism available in systems programs and general purpose computations. Floating-point calculations and heavily iterated loops are less important in general purpose code than in scientific applications. In scientific applications the critical factor which determines execution time is the interval between the start of successive loop iterations, whereas in general purpose computations the major barrier to improved instruction throughput is the limited parallelism available within basic code blocks.

Other investigations [5-8] suggest that the potential for parallelism within basic blocks is limited to an average of no more than two instructions per execution cycle. Parallelism is limited by the data dependencies between instructions and by the short length of basic blocks. As a result HARP aims to improve performance by minimising the latency between pairs of dependent instructions. Furthermore, the HARP compiler increases the potential for parallel instruction execution by overlapping the execution of instructions from distinct basic blocks and merging separate basic blocks into single units.

Work at Hatfield is being carried out using a small research compiler which takes a simple block structured language with Modula-2 syntax and compiles it into HARP code. The compiler output is verified using a simulation of the HARP model written in the hardware description language, ELLA [9]. Results from a variety of benchmarks are being used to finalise the specification of a HARP VLSI processor chip containing four pipelines, which is being designed in collaboration with the Division of Electrical Engineering.

3. Summary of the Prototype Chip (iHARP)

In contrast to most RISC pipelines there is no load delay. The use of an ORed indexing addressing mechanism [10,11] allows data loaded from the data cache in one instruction to be used directly as an operand in an immediately following one. It also allows the number of pipeline stages to be reduced from five to four. In HARP conventional condition codes are replaced by a group of Boolean flags or registers which record the results of specific comparisons. Branch instructions then test one of the Boolean registers.

3.1 The ORed Indexing Addressing Mechanism

HARP has two addressing modes, register indirect plus index, and register indirect plus offset. The address units compute memory addresses by ORing the two address components. In order to use this mechanism, the compiler must ensure that the OR operation is equivalent to an addition. For example, to access a local variable on the stack an offset must be added to the stack pointer.

```
LD      R20, offset (SP)
```

The compiler attempts to ensure that the stack pointer always contains a multiple of a power of two and that the offset is always less than that power of two. As long as this is the case the stack pointer and offset will never have any non-zero bits in common and so can be safely ORed together to achieve an addition.

3.2 The Instruction Execution Pipeline

Data dependencies between instructions constitute a major obstacle to high instruction throughput in all architectures.

HARP therefore uses a steamlined four-stage pipeline and unrestricted register bypassing to allow operands computed or loaded by one long instruction to be used by instructions in the immediately following long instruction. A two instruction branch architecture is also used to reduce the branch delay to one cycle.

The ORed indexing addressing mechanism allows memory addresses to be made available at the end of the Instruction Decode stage of the pipeline and so allows the ALU and Memory Reference operations, which often represent two stages of a pipeline, to be combined into one stage. Hence HARP has the following four stage pipeline [12].

IF	Fetch long instruction from instruction cache
RF	Instruction decode & Register Fetch Calculate branch addresses in PC unit OR memory address components in address units
ALU / MEM	ALU operation or Boolean operation Memory Reference
WB	Write Back

Five stage pipelines, such as the MIPS-X pipeline [13,14], incur a load delay of one cycle. Other four stage pipelines have a two-cycle Load with the address being calculated in one cycle and the data accessed in the second cycle. The HARP pipeline has a one-cycle Load instruction (providing a cache hit occurs), and it eliminates the load delay as the following diagram illustrates.

```
IF      RF      ALU/MEM     WB
                            --> data available
No Delay Slot.              here and bypassed
                            to -->
        IF      RF          ALU/MEM         WB
```

3.3 Boolean Registers and Branching

Boolean registers remove the resource bottleneck of a single condition code register and increase the opportunities for producing parallel code. The Boolean registers allow all instructions to be conditionally executed, with each short instruction independently testing one of the Boolean registers. Testing B0 - which contains a value representing logical FALSE - is used to ensure sequential instruction execution when required. Conditional execution allows basic blocks to be merged and considerably eases the problem of filling branch delay slots.

Branch instructions test a specific Boolean register. An unconditional branch, BRA, is provided at assembly language level and implemented by testing B0. A branch to subroutine instruction, which saves the return address in a general purpose register on procedure entry, is also provided. A 'move register to PC' instruction reverses this operation on procedure exit.

Branch instructions which cause a pipeline to stall or be flushed have a major impact on pipeline efficiency. The cost of a branch instruction, in terms of the total number of cycles taken, depends on the choice of branch architecture and the pipeline structure [15]. In a two instruction branch sequence the result of a test or compare instruction is stored in a register, or in flags, and the result is then used by a separate branch instruction. Single instruction branches specify the operands to be compared, the branch condition and the branch address in one instruction.

HARP uses a two instruction branch mechanism. Relational or Boolean instructions return a value to a Boolean register which is then tested by a subsequent branch instruction. Using a

two instruction branch sequence results in a branch delay of one cycle.

The two instruction branch sequence was chosen because

- using two instructions rather than one is less significant in a parallel architecture,
- a separate compare instruction can be scheduled independently in any instruction slot preceding the branch,
- the result of the comparison is made explicitly available in a Boolean register.

The final point is crucial in HARP since the Boolean registers are also used to provide conditional instruction execution.

4. Conditional Execution of Instructions

HARP does not have conventional condition codes but instead provides a set of one-bit Boolean registers. The contents of these registers are directly manipulated by the Boolean instructions and are also tested by the conditional branch instructions. The Boolean registers are used to implement conditional instruction execution. All short instructions, including the conditional branch instructions and the jump to subroutine instruction, may be conditionally executed. Execution depends on the value contained in the Boolean register specified in the instruction. For example

```
TB1  SUB  R15, R16, R17   * if Boolean register B1 contains the value TRUE
                          * R15 := R16 - R17
```

HARP's instruction timings [4] allow a Boolean value computed in one instruction to control the execution of an immediately following instruction.

While the appropriate unit in computational, relational and Boolean instructions will always compute a result during the execution of the instruction, the final register write back cycle is only enabled if the execution condition is met. In the case of a memory access instruction, a data cache cycle is not initiated if the execution condition fails. Finally, in the case of a branch instruction both the execution condition and the branch condition must be true for the branch to be taken.

HARP's conditional instruction execution facility is a generalisation of a mechanism provided on the Acorn ARM. On ARM all instruction execution is controlled by conventional conditional codes, which are optionally set by computational instructions [16,17].

The compiler uses conditional execution to make full use of the branch delay slot in the compiled code, by moving or copying code from both successor blocks in a conditional branch, and to reduce the number of basic blocks, by conditionally executing both branches of an 'if then else' in parallel.

4.1 Filling the Branch Delay Slot

Conditional execution is used to fill branch delay slots with code from both successor blocks. A major advantage of placing conditionally executed code in the delay slots is that global data flow analysis is no longer required to determine whether instructions can be safely moved between basic blocks. The instructions that are moved or copied into the delay slots are only executed when the attached condition indicates that they would have been executed after the branch instruction.

The principles involved in using conditional execution to produce parallel HARP code can be illustrated by a simple example.

```
Loop:   LD   R2, (SP, R1)        *SP is the stack pointer
        ADD  R1, R1, R2
        GTS  B1, R1, R3          *B1 now contains the branch condition
        BF   B1, Loop            *Branch if B1 is FALSE
        NOP
        LD   R4, (SP, R1)
        ADD  R5, R4, R3
        <next instruction>
```

The code consists of two basic blocks and is only entered at the 'Loop' label. The branch delay slot is indicated by the NOP instruction.

Code can be moved or copied in parallel with the conditional branch instruction itself (BF B1, Loop) and into the branch delay slot by making it conditional on the branch condition - namely the value in Boolean register B1.

If the value in B1 is TRUE then the sequential code will be executed so these two lines of code can be moved in parallel on the condition TB1 (TRUE B1).

```
Loop:   LD   R2, (SP, R1)
        ADD  R1, R1, R2
        GTS  B1, R1, R3
        BF   B1, Loop;     TB1 LD   R4, (SP, R1)
                           TB1 ADD  R5, R4, R3
        <next instruction>
```

If the value in B1 is FALSE then the branch will be taken so the code from the branch target can be copied in parallel on the condition FB1 (FALSE B1).

```
Loop:   LD   R2, (SP, R1)
        ADD  R1, R1, R2
Loop+2: GTS  B1, R1, R3
        BF   B1, Loop+2; TB1 LD   R4, (SP, R1); FB1 LD   R2, (SP, R1)
                         TB1 ADD  R5, R4, R3;   FB1 ADD  R1, R1, R2
        <next instruction>
```

Note that the code concerned was copied rather than moved since the code fragment shown is entered at the 'Loop' label. In practice this label is often the target of another jump and the two lines of code could end up being moved out to another branch slot.

The original code contains seven lines of sequential instructions while the final code contains five lines of parallel instructions. However, the loop itself has been reduced from five sequential short instructions to three long instruction words. The final code still consists of two basic blocks. However, if the top two lines are moved into another block then the final code fragment would consist of just one basic block.

4.2 Reducing the Number of Basic Blocks

The reduction in the number of basic blocks achieved by conditionally executing a piece of code involving an 'if the else' can be seen by considering the following code fragment.

```
        GTS  B1, R1, R2
        BF   B1, Else
        NOP
        LD   R3, (SP, R1)
        ADD  R1, R1, R3
        BRA  Out
        NOP
Else:   LD   R4, (SP, R2)
        ADD  R2, R2, R4
Out:    <next instruction>
```

When conditional execution is used to overlap the basic blocks the code produced is considerably reduced, partly because the unconditional branch is no longer required.

```
GTS  B1, R1, R2
TB1  LD   R3, (SP, R1); FB1 LD   R4, (SP, R2)
TB1  ADD  R1, R1, R3;   FB1 ADD  R2, R2, R4
<next instruction>
```

While the original code had four basic blocks, the final code has only one. Furthermore, the final code is a fragment embedded in a larger basic block which may now yield further parallelism.

5. Conditional Execution versus Boosting

Smith, Lam and Horowitz [1] introduce the concept of 'boosting', whereby instructions from one of the possible instruction streams following a conditional branch are scheduled by the compiler for execution in the basic block containing the branch itself. The values computed by these instructions are put into shadow registers until the result of the branch is known. If the result of the branch is successfully predicted the shadow values are then copied into the appropriate register in the sequential register file. If the prediction is incorrect the results are squashed.

The use of conditional execution allows both instruction streams following a conditional branch to be scheduled in parallel with the branch and its delay slot. Consequently there is a greater potential for reducing the number of long instructions in a program than could be achieved by 'boosting' a single instruction stream. 'Boosting', on the other hand, allows code from the chosen instruction path to be moved further up in the basic block than does the method described in this paper. Our experience has been, though, that basic blocks are quite short - especially after parallelising the code using conditional execution - so that very little would be gained by this.

In any larger basic blocks boosting could allow code to be moved substantially ahead of both the compare and branch instructions. In HARP, since the result of a compare instruction is held in a Boolean register, the compare instruction itself could be moved up in the basic block away from the branch instruction. Consequently code could be conditionally executed in parallel with all the instructions following the compare instruction and not just in parallel with the branch instruction and its delay slot.

The 'boosting' scheme of Smith et al requires the use of profiling to determine the most-likely branch path through the program, or some other method of individual branch prediction. If the branch prediction is incorrect then no advantage is gained. In HARP there is no need to perform any profiling and the advantages are there whichever branch path is taken.

If profiling is used to predict the outcome of a number of branches in succession, then the

boosting scheme can be extended to allow code movement through multiple blocks in a manner similar to Fisher's trace scheduling [18]. This could not be achieved in HARP, though the code that is produced by HARP is more compact than that due to boosting.

Finally, HARP only requires a set of single bit Boolean registers to implement conditional execution, whereas the boosting scheme requires the duplication of the complete register set.

6. Conclusions

This paper has described how the distinctive features of HARP are combined with an optimising compiler to increase instruction throughput and to utilise the low-level parallelism available in general purpose code.

Section three of this paper described the features adopted in HARP to increase instruction throughput by reducing instruction latency, such as the ORed indexing addressing mechanism, the compact four-stage pipeline, unrestricted register bypassing and a single branch delay slot. It also introduced the concept of multiple Boolean registers which are used with great effect to maximise the parallel execution of code by allowing the independent conditional execution of all short instructions.

Section four described how conditional execution was utilised and illustrated this with examples. Conditional execution allows the compiler to combine basic blocks and increase the amount of realisable parallelism. Conditional execution is also fundamental to a delayed branch scheme which overlaps the execution of instructions from distinct basic blocks. The global data flow analysis that is needed in order to move instructions between basic blocks in other implementations is not needed with this scheme. Nor is there any need to perform any sort of profiling to determine the most-likely branch path through the program as is needed in other schemes.

The success of the project ultimately depends on how successfully the compiler schedules instructions in parallel. Initial results are encouraging. Short benchmarks suggest that instruction scheduling will reduce the instruction count of sequential HARP code by about 55%. This figure corresponds to a reduction of about 45%, if it is assumed that all the branch delay slots have already been filled. These results require an average of about two short instruction to be scheduled in every long instruction and suggest that it is realistic to fabricate a HARP processor chip capable of achieving a sustained instruction execution rate significantly in excess of one instruction per cycle.

Acknowledgements

The authors gratefully acknowledge the support of the other members of the HARP team. In particular they would like to thank Sue Gray for her major contribution to the whole HARP project, Gordon Green for his work on the ELLA simulation and Paul Findlay, Brian Johnson and Simon Trainis for their work on the design of the HARP processor chip.

They would also like to thank Dr. S. L. Stott, Professor L.C.W. Dixon and J. A. Davis for their support throughout the HARP project.

The HARP project is supported by S.E.R.C. Research Grant GR/F 88018. Part of this work was also supported by a S.E.R.C. Research Studentship.

References

[1] M. D. Smith, M. S. Lam and M. A. Horowitz, Boosting Beyond Static Scheduling in a Superscalar Processor, *Proc. 17th Ann. Int. Symp. Computer Architecture* (1990) 344-354.

[2] G. B. Steven, S. M. Gray and R. G. Adams, HARP: A Parallel Pipelined RISC Processor, *Microprocessors and Microsystems* , Vol 13, No 9 (November 1989) 579-587.

[3] R. G. Adams, S. M. Gray and G. B. Steven, Utilising Low Level Parallelism in General Purpose Code: The HARP Project, *to appear in Microprocessing and Microprogramming*.

[4] G. B. Steven and S. M. Gray, Specification of a Machine Model for the HARP Architecture and Instruction Set, Version 2, *Computer Science Technical Report*, Hatfield Polytechnic (1990) .

[5] W. W. Hue and P. P. Chang, Exploiting Parallel Microprocessor Microarchitectures with a Compiler Code Generator, *Proc. 15th Ann. Int. Symp. Computer Architecture,* (June 1988) 45-53.

[6] C. C. Foster and E. M. Riseman, The Inhibition of Potential Parallelism by Conditional Jumps, *IEEE Trans. Comput.* , Vol C-21, (December 1972) 1405-1411.

[7] C. C. Foster and E. M. Riseman, Percolation of Code to Enhance Parallel Dispatching and Execution, *IEEE Trans. Comput.*, Vol C-21, (December 1972) 1411-1415.

[8] A. R. Pleszkun and G. S. Sohi, The Performance Potential of Multiple Functional Unit Processors, *Proc. 15th Ann. Int. Symp. Computer Architecture*, (June 1988) 37-44.

[9] G. J. Green, Simulation of the HARP Architecture in ELLA, *Computer Science Technical Report*, Hatfield Polytechnic (1990).

[10] G. B. Steven, A Novel Effective Address Calculation Mechanism for RISC Microprocessors, *ACM Computer Architecture News* , Vol 16, No 4 (September 1988) 150-156.

[11] A. Glew, ORed Indexing, *Unpublished Communication from Author,* (April 1989).

[12] S. M. Gray, Considerations in the Design of an Instruction Pipeline for a Reduced Instruction Set Computer, *Computer Science Technical Report,* No 83, Hatfield Polytechnic (1988).

[13] P. Chow, MIPS-X Instruction Set and Programmer's Manual, *Technical Report* No CSL-86-289, Computer Systems Laboratory, Stanford University, California, USA (May 1986).

[14] M. Horowitz, P. Chow, D. Stark, R. T. Simoni, A. Salz, S. Przybylski, J. Hennessy, G. Gulak, A. Agarwal and J. M. Acken, MIPS-X: A 20-MIPS Peak, 32-bit Microprocessor with On-Chip Cache, *IEEE J. Solid State Circuits* , Vol SC-22, No 5 (October 1987) 790-799.

[15] S. McFarling and J. Hennessy, Reducing the Cost of Branches, *Proc. 13th Ann. Int. Symp. Computer Architecture* , (June 1986) 396-403.

[16] S. Furber, VLSI RISC Architecture and Organization, Marcel Dekker Inc., New York, (1989).

[17] VL86C010 32-Bit RISC MPU and Peripherals Users Manual, VLSI Technology Inc, Prentice-Hall, New Jersey (1989).

[18] J. A. Fisher, Very Long Instruction Word Architectures and the ELI-512, *Proc. 10th Ann. Int. Symp. Computer Architecture* (1983) 140-150.