DIVISION OF COMPUTER SCIENCE

HARP: A Statically Scheduled Multiple-Instruction-Issue Architecture and its Compiler

(Submitted to 2nd Euromicro Workshop on Parallel and Distributed Processing, Malaga,Spain, January 1994)

R. G. Adams
S. M. Gray
G. B. Steven

Technical Report No.163

September 1993

HARP: A Statically Scheduled Multiple-Instruction-Issue Architecture and its Compiler

R. G. Adams, S. M. Gray and G. B. Steven
Division of Computer Science,
The University of Hertfordshire,
College Lane, Hatfield, Herts AL10 9AB, UK

Abstract

This paper presents the results of an investigation into the performance of a new statically scheduled multiple-instruction-issue architecture and its compiler. HARP is a Long Instruction Word Architecture developed in conjunction with a simple compile-time scheduling technique called conditional compaction. The architecture is characterised by a conditional execution mechanism which is used by the scheduler to pack the instructions within a procedure into long instruction words. The study compares the speedups obtained for the C and Modula-2 versions of a set of short, general purpose, integer benchmarks, running on simulations of the architecture with different functional unit configurations.

1.0 Introduction

Multiple-instruction-issue machines utilise the instruction level parallelism available in compiled code to increase processor performance. Scalar RISC processors use compiler optimisations and a single, streamlined instruction pipeline to achieve execution rates approaching the upper limit of one instruction per cycle.[1] Multiple-instruction-issue machines[2] provide multiple pipelined functional units in order to fetch, decode and execute several instructions per cycle. Multiple-instruction-issue processors can be divided into two categories: superscalar processors, which provide hardware for the run-time detection of parallelism,[3] and very long instruction word (VLIW) machines, which rely on the compiler to schedule concurrent instructions into very long instruction words[4,5].

The objective of the HARP (Hatfield Advanced RISC Processor) project[6,7] is to develop a VLIW processor/compiler system which will achieve sustained execution rates in excess of two instructions per cycle for general-purpose code. The project began with the specification of a machine model[8] and then continued with the development of compiler systems[9,10] and the design and testing of iHARP, a VLSI integrated circuit implementation of the machine model.[11,12]

This paper presents the essential features of HARP machine model and describes the conditional compaction scheduling technique. It then gives the results of experiments to compare the performance of the technique, for C and Modula-2 versions of the benchmarks, when a maximum of four ALU and one Boolean instruction are scheduled in parallel with a variable number of branch and memory reference instructions.

2.0 The Machine Model

The machine model[8] describes a class of RISC architectures with a variable number of instruction pipelines. Multiple ALUs, a Boolean Unit, a PC unit and a maximum of two address units allow several ALU operations, one Boolean operation and a maximum of two memory reference and two branch instructions to be executed in parallel. The compiler translates source programs into short instructions which specify typical RISC operations; the instruction scheduler then selects short instructions which can be executed in parallel and packs them into long instruction words (LIWs). The model fetches LIWs, one per cycle, from an instruction cache and passes the component short instructions through the multiple pipelines.

Conventional condition codes are replaced by a set of one-bit Boolean registers which are set explicitly by relational or Boolean instructions. The Boolean registers are tested by conditional branch instructions, and are also used to control instruction execution. For example, the instruction T B2 ADD R1,R2,R3 will only be executed if the Boolean register B2 holds the value TRUE.

The model provides 64 general-purpose registers, R0-R63 (R0 is hardwired to 0), 32 Boolean registers, B0-B31, and two addressing modes: register indirect with index, and register indirect with displacement.

2.1 Instruction Latency

The HARP processor is specifically targeted at general-purpose applications. Such programs are likely to contain relatively low amounts of instruction level parallelism compared with numeric applications. It is therefore essential not to squander any of the parallelism, gained by the compiler, on increased instruction latencies. This section discusses features of the HARP architecture which have a major impact on instruction latency: the instruction pipeline and full register bypassing, ORed addressing, and the use of Boolean registers.

All HARP instructions are executed in the following four stage pipeline:

IF	Fetch next instruction from instruction cache
RF	Fetch register operands from register file
ALU/MEM	Perform operation or access data cache
WB	Return results to register file

A computational instruction uses all four stages: reading two operands in the second stage, performing the computation in the third stage and returning a result to a register in the final stage. The resulting computational delay of 2 cycles is removed by the provision of 32-bit bypass paths from all ALU outputs to all ALU inputs which allows the immediate re-use of data. A relational instruction uses one of the dedicated high-speed comparators provided in each pipeline to compute a Boolean value during the first half of the ALU/MEM stage. This result can then be bypassed to the RF stage of the next instruction for control purposes. A branch instruction tests the branch condition and computes the branch target (using a dedicated adder) in the RF stage of the pipeline.This timing results in a branch delay of one cycle. A memory reference instruction computes the memory address in the RF stage, and accesses a separate off-chip data cache in the ALU/MEM stage. Data loaded from the data cache can then be bypassed directly to the ALU inputs of the next instruction.This timing has the advantage of removing any load delay (recent simulations[2] suggest that introducing a one cycle load delay in HARP would degrade the instruction-issue rate by approximately 25%) but relies on the processor's ability to compute the memory addresses in the RF stage of the pipeline. However, since there is not time to access the register file and perform a 32-bit addition of the address components in a single cycle, some simplification of the addressing mechanism is essential.

HARP implements a distinctive ORed addressing mechanism, where a bitwise OR operation is performed between the two address components to calculate the effective address.[13] This simple mechanism is equivalent to an addition, provided there is never a carry in the addition (i.e. no two bits in the same position are both set to 1). HARP compilers enforce this requirement by starting all procedure activation records on a power-of-two boundary.[9] The least significant bits of the stack pointer are forced to zero on procedure entry and variables are accessed relative to the stack pointer using ORed indexing. The power-of-two boundary used is adjusted from procedure to procedure to avoid excessive memory fragmentation. Effectively a new variable-sized 'page' is allocated on the stack on procedure entry.

Finally HARP's conditional execution mechanism can be viewed as a mechanism for reducing branch latency. At run-time a conditionally executed instruction is only allowed to change the machine state if its execution condition is true. Instructions which are normally executed after a branch instruction can be conditionally executed on the value of the branch condition which results in their execution, as soon as the value of the Boolean variable governing the branch is computed.

3.0 Instruction Scheduling

The instruction scheduler takes the output from a sequential HARP compiler and packs the short instructions within each procedure into LIWs. The scheduler is parameterised to produce LIWs which match a particular functional unit configuration of the model. The scheduling process is divided into two phases: local compaction and conditional compaction.

3.1 Local Compaction

Local compaction is used to schedule the short instructions within each basic block into LIWs. Each LIW consists of a fixed number of branch, ALU, memory reference and Boolean instruction slots. Slots which are not filled by the scheduler are packed with NOPs.

For each basic block the local compaction program builds a directed acyclic graph (DAG) to represent the partial ordering of short instructions which the scheduler maintains in order to preserve data integrity.[14] Dependencies between data held in registers are detected by comparing the input and output registers used by pairs of instructions. Memory reference disambiguation is only performed between pairs of instructions which both use register indirect with offset addressing relative to the stack pointer, the global pointer or R0 (the value of the stack pointer remains constant throughout the lifetime of a procedure, the values of the global data pointer and R0 remain constant throughout the program). Otherwise store instructions are considered to be data dependent on all preceding load and stores, and load instructions are considered to be data dependent on all preceding stores.

Having computed the DAG, the local compaction program generates an empty LIW, called the current long instruction word (CLIW), and uses the information in the DAG to compute the set of instructions which can be scheduled in the CLIW without violating data integrity (the data available set). List scheduling[15], wherein each instruction is assigned a priority prior to scheduling, is used to determine the order in which the instructions in the data available set are considered for inclusion in the CLIW. A data available instruction can only be scheduled in the CLIW if the CLIW contains enough compatible empty slots, such an instruction is said to be resource available with respect to the CLIW. If the data available instruction currently under consideration is resource available with respect to the CLIW, it is scheduled in the CLIW and the data available set is recalculated. If it is not, the data available instruction with the next highest priority is considered for scheduling in the CLIW. This process is repeated until no more instructions can be scheduled in the CLIW. A new LIW is then generated and the algorithm is repeated until all the non-branch instructions in the block have been scheduled. Finally any remaining branch instruction must, by definition, be scheduled in the penultimate LIW of a compacted block. Hence a branch instruction and its associated NOP are scheduled last, taking into account data and resource dependencies.

3.2 Conditional Compaction

Non-numeric applications are characterised by a high proportion of data dependent branches, small loop bodies, and low loop iteration counts. Hence the conditional compaction technique, which is targeted at general purpose computations, aims to remove the dependencies caused by branch instructions, rather than focusing on the parallelism available within loops.

Conditional compaction is a simple scheduling technique which uses HARP's conditional execution mechanism to move instructions across basic block boundaries. A locally compacted block is conditionally compacted by moving short instructions from its branch destination and sequential successor blocks into the empty slots in its schedule. Instructions which are moved across conditional branches are conditionally executed on the value of the Boolean variable which would result in entry to their native block. This effectively removes the control dependencies caused by branch instructions, and makes global scheduling relatively straightforward.

The blocks in a procedure which are candidates for conditional compaction are held in a list. Initially the ordering of the blocks in the list corresponds to their static ordering in the locally compacted code. The scheduler removes the block from the head of the list, and conditionally compacts this block, which is referred to as the C-block, with each of its successors in the flow graph. Any block which is not already in the list is added to the list, if the conditional compaction process results in the movement of instructions which may permit further compaction to take place. The process of removing and conditionally compacting the block at the head of the list is repeated until the list is empty.

A variation of the local compaction algorithm described above is used to move the non-branch instructions from the C-block's branch destination or sequential successor block into the C-block's locally compacted schedule. If all the non-branch instructions are moved out of the C-block's successor, the scheduler then attempts to move any remaining branch instruction(s) into the C-block's penultimate LIW. If the scheduler succeeds in moving instructions from a successor block which has more than one predecessor, copies of the moved instructions are introduced into the procedure's flow graph to preserve the correctness of the program.

If the C-block ends in an unconditional branch code motion is limited only by data dependencies and the availability of instruction slots in the C-block's locally compacted schedule. If the C-block ends in a conditional branch the scheduler must first verify that the instructions it attempts to move can be conditionally executed on the value of the Boolean variable which governs the C-block's branch. A simple IF ..THEN..ELSE statement will illustrate the effect of conditional compaction:

```
         EQ  B1, R1, R2             /* Calculates a Boolean value */
         BT  B1, ELSE               /* Branch IF Boolean TRUE */
         NOP                        /* Branch delay slot */             Block 1
------------------------------------------------------------------------------
         Instr1          BRA  OUT
         Instr2                                                         Block 2
------------------------------------------------------------------------------
ELSE:    Instr3          Instr4
         Instr5                                                         Block 3
------------------------------------------------------------------------------
OUT:     Instr6
         Instr7          Instr8
         .
         .                                                              Block 4
```

Block 1 branches forwards. In the case of a forward branch the compaction program favours the removal of short branches, by attempting to move instructions from the sequential successor block (block 2) before considering the branch target block (block 3). In this example all the instructions in block 2 can be conditionally executed on the value of B1 which governs the branch. Conditionally executed copies of the non-branch instructions are successfully moved into block 1's schedule, and the corresponding instructions are removed from block 2. Similarly a conditionally executed copy of the remaining branch instruction, F B1 BRA OUT, is then moved into block 1's penultimate LIW (where it is replaced by the equivalent unconditionally executed instruction BF B1,OUT) and the corresponding instruction is removed from block 2. Block 2, which can only be entered from block 1, is then empty, so it is removed from the flow graph. This means that block 1's original branch, BT B1, ELSE, is redundant (as block 3 is now block 1's sequential successor), so it is removed giving:

```
         EQ  B1, R1, R2
         BF  B1, OUT;      F B1 Instr1;
                           F B1 Instr2;                                 Block 1
------------------------------------------------------------------------------
         Instr3            Instr4
         Instr5                                                         Block 3
------------------------------------------------------------------------------
OUT      Instr6
         Instr7            Instr8
         .
         .                                                              Block 4
```

Block 1 has thus acquired a new sequential successor block (block 3) and a new branch target block (block 4), so it is returned to the compaction list where the process is repeated. In this case one instruction is moved from the sequential successor block, and two instructions are moved from the branch target block giving:

```
         EQ  B1, R1, R2
         BF  B1, OUT+2;    F B1 Instr1;   T B1 Instr3;      F B1 Instr6;
                           F B1 Instr2;   F B1 Instr7;                  Block 1
------------------------------------------------------------------------------
         Instr4
         Instr5                                                         Block 3
------------------------------------------------------------------------------
OUT+2    Instr8
         .
         .                                                              Block 4
```

4.0 Investigation and Results

Our investigation was conducted using C and Modula-2 versions of a set of short, general-purpose integer benchmarks, running on a simulation of the machine model. The C programs were translated into sequential HARP assembler by a GNU-CC generated compiler. The Modula-2 versions were translated by an in-house Modula-2 compiler produced as part of the project.[9] The simulator executes fixed length LIWs, containing any combination of short instructions, subject to a maximum of two branch instructions per LIW. Table 1 describes the benchmark programs, and gives the dynamic cycle count for the conditionally compacted sequential code for both sets of benchmarks. Conditional compaction was applied to the sequential code in an attempt to fill the branch delay slots.

Tables 2-5 show the performance of the conditionally compacted code obtained for both sets of benchmarks using configurations of the architecture which allow four ALU and one Boolean instruction to be executed in parallel with one/two branch and one/two memory reference instructions. Table 6 summaries the average speedups over the conditionally compacted sequential code obtained for these configurations. The results of a previous investigation[14] into the performance of the scheduling technique for the C versions of the Stanford integer benchmarks have shown that there is little to be gained by allowing more than four ALU instructions to be executed in parallel.

As can be seen from the results in Table 6, the configuration which allows one branch and one memory reference instruction to be scheduled in parallel achieves speedups of 1.75 and 2.25 for the C and Modula-2 code respectively. Allowing two branches to be scheduled in parallel with a single memory reference increases these speedups by a factor of 2.9% for the C and 4.0% for the Modula-2. (Although it should be noted that since concurrent branch instructions are always executed, or taken, on opposite Boolean conditions only one branch is actually taken). Greater improvements would be obtained by using procedure in-lining to remove subroutine calls which inhibit the performance of the scheduler.[14]

In contrast, allowing two memory reference instructions to be scheduled in parallel with a single branch increases the speedups obtained for the one branch, one memory reference configuration by a factor of 7.4% for the C and 8.4% for the Modula-2. While allowing two branch and two memory reference instructions to be scheduled in parallel achieves a speedup of 1.95 for the C, and 2.55 for the Modula-2. These results represent increases of 11.43% and 13.3% over the single branch, single memory reference configuration and imply that the hardware required to executed two parallel memory reference instructions, principally a greater data cache bandwidth and a dual ported data memory, is well worth considering.

Finally it can be seen that in all cases the speedups obtained for the Modula-2 programs are significantly greater than the speedups obtained for the C. This is due in part to differences in the sequential compilers (the GNU_CC generated compiler produces slightly more optimised code) but is mainly a reflection of the differences in the source languages. In particular C does not have array bounds checking; and so translates to shorter basic blocks which have less potential for compaction, particularly local compaction (Table 7 shows that average speedups of 1.44 and 1.97 were obtained for the locally compacted C and Modula-2 code scheduled for the two branch, two memory reference configuration). These results highlight the effect that the benchmark's source language, and the sequential compiler, can have on the performance of the scheduling algorithm, and demonstrate that these factors need to be taken into account when considering the results of any study of this nature.

5.0 Concluding Remarks

This paper presents the machine model for HARP, a LIW architecture characterised by a streamlined four stage instruction pipeline, unrestricted register bypassing, an ORed addressing mechanism, and conditional instruction execution. It then describes conditional compaction; a simple instruction scheduling technique which uses HARP's conditional execution mechanism to increase the scope of the scheduler beyond basic blocks. The paper gives the results of experiments to compare the speedups of C and Modula-2 versions of a set of benchmark programs, running on a simulation of the machine model which allows a maximum of one Boolean and 4 ALU instructions to be scheduled in parallel with one/two branch instructions and one/two memory reference instructions. These experiments result in maximum speedups of 1.95 and 2.55 respectively for the two branch, two memory reference model. These results are in-line with the project objective of developing a processor-compiler system capable of sustained execution rates in excess of two instructions per cycle for general-purpose code.

References

1 **Hennessy, J and Patterson, D A** *Computer Architecture a Quantitative Approach*, Morgan Kaufmann, San Mateo, California (1991)

2 **Chang, P P, Mahlke, S A, Chen, W Y, Warter, N J & Hwu, W W** 'IMPACT: An Architectural Framework for Multiple-Instruction-Issue Processors' *Proc. 18th Ann. Int. Symp. Computer Architecture* (May 1991) pp 266-275

3 **Johnson, M** *Superscalar Microprocessor Design*, Prentice Hall (1991)

4 **Colwell, R P, Nix, R P, O'Donnell, J J, Papworth D B & Rodman, P K** 'A VLIW Architecture for a Trace Scheduling Compiler' *IEEE Trans. Comput.* Vol 37 No 8 (Aug. 1988) pp 967-979

5 **Labrousse, J & Slavenburg, G A** 'CREATE-LIFE: A Modular Design Approach for High Performance ASIC's' *Proc. IEEE COMPCON* (Spring 1990) pp 427-433

6 **Steven, G B, Gray, S M & Adams, R G** 'HARP: A Parallel Pipelined RISC Processor' *Microprocessors and Microsystems* Vol 13 No 9 (Nov. 1989) pp 579-587

7 **Adams, R G, Gray, S M & Steven, G B** 'Utilising Low Level Parallelism in General Purpose Code: The HARP Project' *Microprocessing and Microprogramming* Vol 29 No 3 (Oct. 1990) pp 137-149

8 **Steven, G B & Gray, S M** 'Specification of a Machine Model for the HARP Architecture and Instruction Set - Version 3' *Comp.Sci.Tech.Rep.No 117* Hatfield Poly. UK (Jan. 1991)

9 **Gray, S M** 'Code Generation for a Long Instruction Word Architecture' *PhD Thesis* Hatfield Poly. UK (Dec. 1991)

10 **Wang, L** 'Instruction Scheduling for a Family of Multiple-Instruction-Issue Architectures' *PhD Thesis* The University of Hertfordshire, UK (Expected Sept. 1993)

11 **Findlay, P A, Trainis, S A, Steven, G B & Adams, R G** 'HARP: A VLIW RISC Processor' *CompEuro91* Bologna (May 1991) pp 368-372

12 **Steven, G B, Adams, R G, Findlay, P A & Trainis, S A** 'iHARP: A Multiple-Instruction-Issue Processor' *IEE Part E Comput. and Digital Techniques* (to appear)

13 **Steven, G B** 'A Novel Effective Address Calculation Mechanism for RISC Microprocessors' *ACM Computer Architecture News* Vol 16 No 4 (Sept. 1988) pp 150-156

14 **Gray, S M & Adams, R G** 'Using Conditional Execution to Exploit Instruction Level Concurrency' submitted for publication *Software Practice and Experience*

15 **Davidson, S, Landskov, D, Shriver, B & Mallett, P W** 'Some Experiments in Local Microcode Compaction for Horizontal Machines' *IEEE Trans. Comput.* Vol C-30 No 7 (Jul. 1981) pp 460-477

Program	Dynamic Instr. Count for Cond.Compacted Seq. Code		Description
	C	Modula-2	
Bubble	1315	2702	Bubble sort of 10 integers
Quick	1121	1937	Recursive quick sort of 10 integers
Perm	20013	29251	Recursive computation of all permutations of 5 elements
Queens	25067	49720	Recursive solution of the 8 queens chess problem
Intmm	3698	7419	Multiplies two 5x5 integer matrices
Binsearch	176	222	Iterative binary search of an array of 9 integers
Fib	1050	1207	Recursive computation of the first 6 Fibonacci nos.
Sieve	6034	8301	Computes the prime numbers between 3 and 43 using the Sieve of Eratosthenes

Table 1. The Benchmark Programs

Program	Dynamic Instruction Count		Speedup Over Conditionally Compacted Sequential Code	
	C	Modula-2	C	Modula-2
Bubble	634	1227	2.07	2.20
Quick	785	1061	1.43	1.83
Perm	12808	14192	1.56	2.06
Queens	16215	18034	1.55	2.76
Intmm	1808	2538	2.05	2.92
Binsearch	90	97	1.96	2.29
Fib	747	708	1.41	1.70
Sieve	3056	3767	1.97	2.20
Average			1.75	2.25

Table 2. Performance of the Conditionally Compacted Code for the 1 branch, 1 memory reference, 4 ALU, 1 Boolean Configuration of the Model

Program	Dynamic Instruction Count		Speedup Over Conditionally Compacted Sequential Code	
	C	Modula-2	C	Modula-2
Bubble	632	1213	2.08	2.23
Quick	739	1041	1.52	1.86
Perm	12460	13700	1.61	2.14
Queens	15989	18017	1.57	2.76
Intmm	1754	2429	2.11	3.05
Binsearch	86	86	2.05	2.58
Fib	703	708	1.49	1.70
Sieve	3054	3415	1.98	2.43
Average			1.80	2.34

Table 3. Performance of the Conditionally Compacted Code for the 2 branch, 1 memory reference, 4 ALU, 1 Boolean Configuration of the Model

Program	Dynamic Instruction Count		Speedup Over Conditionally Compacted Sequential Code	
	C	Modula-2	C	Modula-2
Bubble	608	1217	2.16	2.22
Quick	727	1001	1.54	1.94
Perm	9858	11414	2.03	2.56
Queens	14972	17131	1.67	2.90
Intmm	1803	2506	2.05	2.96
Binsearch	89	94	1.98	2.36
Fib	634	513	1.66	2.35
Sieve	3056	3767	1.97	2.20
Average			1.88	2.44

Table 4. Performance of the Conditionally Compacted Code for the 1 branch, 2 memory reference, 4 ALU, 1 Boolean Configuration of the Model

Program	Dynamic Instruction Count		Speedup Over Conditionally Compacted Sequential Code	
	C	Modula-2	C	Modula-2
Bubble	606	1203	2.17	2.25
Quick	681	981	1.65	1.97
Perm	9510	10684	2.10	2.74
Queens	14746	17002	1.70	2.92
Intmm	1749	2397	2.11	3.10
Binsearch	85	83	2.07	2.67
Fib	590	513	1.78	2.35
Sieve	3054	3415	1.98	2.43
Average			1.95	2.55

Table 5. Performance of the Conditionally Compacted Code for the 2 branch, 2 memory reference, 4 ALU, 1 Boolean Configuration of the Model

Max no. of branch and mem. ref. instrs. scheduled in parallel with 4 ALU and 1 Bool instr.	Average speedup over the conditionally compacted sequential code.	
	C	Modula -2
1 branch, 1 memory ref.	1.75	2.25
2 branch, 1 memory ref.	1.80	2.34
1 branch, 2 memory ref.	1.88	2.44
2 branch, 2 memory ref.	1.95	2.55

Table 6. Average Speedups of the Conditionally Compacted Benchmarks for Different Configurations of the Machine Model.

Program	Dynamic Instruction Count		Speedup Over Conditionally Compacted Sequential Code	
	C	Modula-2	C	Modula-2
Bubble	1075	1510	1.22	1.79
Quick	936	1209	1.20	1.60
Perm	10928	11091	1.83	2.64
Queens	21276	21526	1.18	2.31
Intmm	2116	3261	1.75	2.28
Binsearch	118	135	1.49	1.64
Fib	681	584	1.54	2.07
Sieve	4656	5696	1.30	1.46
Average			1.44	1.97

Table 7. Performance of the Locally Compacted Code