Static Instruction Scheduling for the HARP Multiple-Instruction-Issue Architecture

S. M. Gray, R. G. Adams, G. J. Green and G. B. Steven

October 1992

School of Information Sciences
Division of Computer Science
Technical Report No.142

Static Instruction Scheduling for the HARP Multiple-Instruction-Issue Architecture

S. M. Gray, R. G. Adams, G. J. Green and G. B. Steven

Division of Computer Science,

The University of Hertfordshire

College Lane, Hatfield, Herts AL10 9AB, UK

ABSTRACT

HARP is a new multiple-instruction-issue architecture developed at The University of Hertfordshire. This paper describes the essential features of the HARP machine model and presents two compile-time scheduling techniques, called local and conditional compaction, which have been developed for the architecture.

Local compaction schedules the instructions within a basic block. Conditional compaction uses HARP's conditional execution mechanism to schedule instructions across basic block boundaries. This paper reports performance measurements obtained using simulations of the model. These results indicate that a HARP processor will achieve sustained instruction execution rates in excess of two sequential instructions per cycle for compiled, integer, general-purpose computations.

INTRODUCTION

Scalar RISC processors use efficient instruction pipelining and compiler optimisations to achieve instruction execution rates approaching the upper limit of one instruction per cycle[1]. Hence there is now a growing interest in machines which exploit instruction level concurrency to boost performance beyond the one instruction per cycle barrier. Processors which provide parallel pipelined functional units in order to fetch, decode and execute several instructions per cycle are generally referred to in the literature as multiple-instruction-issue machines[2]. Multiple-instruction-issue processors can be divided into two categories: superscalar processors which provide hardware for the run-time detection of parallelism[3-5] and VLIW machines which rely on the compiler to schedule concurrent instructions into very long instruction words[6-8].

Many of the early multiple-instruction-issue machines[5-7,9-11] are targeted at scientific code. In contrast HARP[12-13] is one of several more recent projects[14-15] which have attempted to exploit the low-level parallelism available in general-purpose computations. Numeric applications are characterised by a high ratio of computations to dynamic branches. Furthermore a large percentage of the branches are do-loop branches which can be resolved early. Hence techniques such as loop unrolling[16] and software pipelining[17], which are targeted at scientific applications, concentrate on exposing the parallelism between successive loop iterations. 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.

The objective of the HARP project is to develop a VLIW processor/compiler system which will achieve sustained instruction execution rates in excess of two instructions per cycle for integer, general-purpose code. The project began with the specification of a machine model[18] and has since developed on two fronts: the first centred around compiler development and the second concerned with the design and testing of iHARP, a

VLSI integrated circuit implementation of the machine model[19-20]. This paper is primarily concerned with the scheduling techniques of local and conditional compaction which have been developed for the architecture[21].

Local compaction schedules the short instructions within a basic block. The local compaction program constructs a directed acyclic graph to represent the partial ordering imposed on the short instructions in a basic block by data dependencies. A first-come-first-served heuristic is then used to list schedule the short instructions in each block into long instruction words. Various studies[15,22] have shown that fine-grained parallelism is limited to a factor of two within basic blocks of general-purpose code. Hence the technique of conditional compaction was developed to extend the scope of scheduling across basic block boundaries. General-purpose programs are characterised by a high proportion of dynamic branches which impede the parallel execution of instructions from before and after the branch. Conditional compaction is a global scheduling technique which uses HARP's conditional execution mechanism to remove the dependencies caused by branch instructions. Conditional execution allows the scheduling of instructions from the sequential and branch destinations without the requirement for global data flow analysis or a branch prediction scheme.

Both these techniques have been implemented in the HRC compiler which generates code for a set of software simulations with different numbers of instruction pipelines[23]. These simulations were used in conjunction with the compiler to obtain performance measurements for a set of short, general-purpose, integer benchmark programs. This paper presents an overview of the essential features of the machine model, describes the local and conditional compaction techniques and presents the results of experiments used to evaluate the combined performance of the architecture and the scheduling techniques.

THE MACHINE MODEL

The machine model[18] 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's sequential component 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. The model fetches one long instruction word per cycle from an instruction cache, and passes the component short instructions through the multiple pipelines (see Figure 1).

The model provides 64, 32-bit, general-purpose registers and 32, 1-bit, Boolean registers. There are five types of short instruction: computational, relational, Boolean, branch and memory reference (see Table 1). The Boolean registers are used to store the results of relational and Boolean instructions, and are tested by conditional branch instructions. Only load and store instructions reference memory, and only two addressing modes are provided: register indirect with index and register indirect with displacement.

In order to achieve HARP's performance objective it is important that the vast majority of short instructions are executed in a single cycle. To this end the model provides several distinctive features specifically designed to minimise the performance penalties due to data dependencies and branch instructions. Minimising the number of wasted cycles is also particularly important for a long instruction word processor where the empty long instruction words, which arise because of result latencies and branch delays, use up some of the available parallelism. Consequently the model uses a compact four-stage instruction pipeline which combined with unrestricted register bypassing and an ORed addressing mechanism ensures that almost all instructions have an operational latency of one cycle.

IF	Fetch long instruction from Icache (Instruction cache)
RF	Instruction decode
	Fetch registers from GP and Boolean register file
	Calculate branch addresses in PC unit
	Calculate memory addresses in address units
ALU / MEM	ALU operation for computational or relational instructions
	Calculate Boolean result in Boolean unit
	Wait for data from memory for a load instruction
	Output data for a store instruction
WB	Write result of computational or load instruction into the general-purpose register file
	Write the result of a relational, Boolean or Boolean load instruction into the Boolean register file

The HARP address units compute memory addresses by ORing the two address components[24]. The compiler guarantees that the OR operation is equivalent to an addition by ensuring that the base register always contains a multiple of a power of two and that the offset is always less than this power of two. Removing the need for an addition in an address calculation allows memory addresses to be made available at the end of the register fetch stage of the pipeline. ALU operations and memory accesses can then be carried out in the same pipeline stage. Compute delays and load delays are removed by bypassing the result of the ALU/MEM stage of a short instruction directly to the ALU/MEM stage of the next instruction.

The model uses a two instruction branch mechanism which results in a branch delay of one cycle. The result of an instruction which performs a relational or Boolean operation is stored in a Boolean register. This result is then tested by a subsequent branch instruction. Using a set of single bit Boolean registers to store the results of comparisons avoids the resource bottleneck of a single set of condition codes and thus increases the potential for parallelism.

Parallel execution is supported by the multiple functional units and a conditional execution mechanism. All short instructions may be conditionally executed on the value of a Boolean register specified in the instruction. For example the instruction

```
F B3  SUB R21, R20, #3
```

is executed if and only if the Boolean register B3 contains the value FALSE. Conditional execution is fundamental to the technique of conditional compaction.

INSTRUCTION SCHEDULING

The HRC compiler consists of two components: the sequential component and the instruction scheduler. The sequential component translates a subset of Modula-2 into unconditionally executed short instructions. The instruction scheduler schedules short instructions onto the HARP pipelines by packing them into long instruction words. The compiler is parameterised to produce code for a set of software simulations with a variable number of instruction pipelines. In HRC the compaction process is divided into two phases: first local compaction is used to schedule the unconditionally executed short instructions within each basic block into long instruction words, then conditional compaction is used to schedule conditionally executed instructions from each block's successors in the flow graph into the existing schedule for the locally compacted block.

Local Compaction

There are three types of dependencies between instructions which determine whether they can be scheduled in parallel: data dependencies, resource dependencies and control dependencies.

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. Given two short instructions s_i and s_j, where s_i precedes s_j in the sequential code, this partial ordering is expressed informally in Table 2. Parallel execution is possible in the case of a register use-definition dependency as the old value of the data used by s_i is read from the register file in the register fetch stage of the pipeline, one cycle before the new value is computed by s_j in the ALU stage. The DAG is constructed by scanning backwards through the block comparing each instruction to each of its predecessors. Register dependencies are detected by comparing the input and output registers of each pair of instructions. No attempt is made to disambiguate memory references. The scheduler simply assumes that all memory addresses are potentially equivalent; and thus maintains the order of store instructions with respect to other loads and stores.

In the HARP model each slot in a long instruction word is dedicated to a particular pipeline. Hence potential resource conflicts over functional units are detected by comparing the set of slots in which a short instruction may be scheduled with the set of empty slots in a long instruction word.

Finally branch instructions which determine a program's flow of control must, by definition, be scheduled in the penultimate long instruction word of a compacted block.

Hence a branch instruction and its associated NOP are scheduled last, taking into account data and resource dependencies.

The local compaction program builds a list of long instruction words, one long instruction at a time, in sequential order, using the following algorithm:

```
WHILE there are still instructions to be scheduled (excluding a branch and NOP) DO
    Generate the current long instruction word (CLIW)
    Compute data available set
    REPEAT
        Find the instruction in the data available set, with the highest priority, which
        will fit into the CLIW
            IF such an instruction exists THEN
                Schedule the instruction in the CLIW
                Update the data available set
            END
    UNTIL No more instructions can be scheduled in the CLIW
    Add the CLIW to the long instruction word list
END
Schedule the branch and NOP
```

To form the current long instruction word (CLIW) the program generates an empty long instruction and uses the information in the DAG to compute the set of short instructions which can be scheduled in the CLIW without violating data integrity. These instructions are referred to as the data available set. Since two short instructions cannot be scheduled in the same long instruction word if they both require the same slot, each short instruction in the block is assigned a priority. The scheduler uses a first-come-first-served heuristic, where the priority of an instruction reflects its position in the sequential code. The scheduler selects the instruction in the data available set with the highest priority which does not have a slot conflict with any instruction already in the CLIW. If such an instruction exists it is scheduled in the CLIW. This process is referred to as list scheduling[25]. Since the HARP pipeline allows two short instructions which have a register use-definition dependency to be scheduled in parallel, the local compaction program updates the available set of instructions each time it adds a new short instruction to the CLIW. The process of scheduling the instruction with the highest priority and updating the available set is repeated until no more short instructions from the available set can be scheduled in the CLIW. The CLIW is then added to the long instruction word list, a new long instruction word is generated and the whole process is repeated until all the short instructions in the block have been scheduled. Finally the branch instruction and its associated NOP are scheduled subject to data and resource dependencies.

Conditional Compaction

Conditional compaction is a scheduling technique developed to increase the performance of the HARP architecture by extending the scope of the scheduler beyond basic blocks. Conditional compaction is targeted at non-numeric applications, and hence seeks to remove the scheduling restrictions imposed by control dependencies, rather than concentrating on utilising the parallelism available between successive loop iterations.

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 the compaction list. The block at the head of this list is selected for conditional compaction. This block is referred to as the C_block. The compaction list is initialised to contain all the blocks in the current procedure. Initially the ordering of the blocks in the list corresponds to their static ordering in the locally compacted code. Thereafter blocks are added to the head of the list. The scheduler removes the C_block from the head of the list, and conditionally compacts the C_block with each of its successors in the flow graph. Any block which has been removed from the compaction list is returned to the list if the conditional compaction process results in changes to its successors which may permit further compaction to take place. The process of removing the C_block from the head of the list and conditionally compacting it with each of its successors in the flow graph is repeated until the compaction list is empty.

If the C_block is terminated by a backward conditional branch the scheduler attempts to schedule instructions from the branch target block before considering the sequential successor block. This should favour the compaction of code within a loop. If the C_block is terminated by a forward conditional branch the scheduler attempts to schedule instructions from the sequential successor block, before considering the branch target block. This should favour the removal of short forward branches wherever possible.

Given a C_block and its successor, the scheduler determines the set of non-branch instructions from the successor block which are candidates for scheduling in the C_block. This set of instructions, which is referred to as the conditionally available set (CASet), is made up of copies of the short instructions from the successor block. Branch instructions are not included in the CASet. If the C_block contains a conditional branch instruction the instructions in the CASet are conditionally executed on the value which results in entry to their native block. Conditional execution places further restrictions on the instructions which are included in the CASet. For example, if the

C_block ends with a conditional branch on B3, any instruction from the successor block which defines B3, or is data dependent on an instruction which defines B3, cannot be conditionally executed on the value of B3 which results in entry to the successor block. More obviously, any instruction in the successor block which is already conditionally executed cannot be included in the CASet.

The scheduler concatenates the short instructions from the C_block and the CASet into a single unit; builds a directed acyclic graph for the unit and uses the DAG to list schedule as many of the instructions from the CASet as possible into the existing schedule for the C_block. The corresponding instructions are then removed from the successor block, and the new, shorter successor is rescheduled (locally compacted). If all the non-branch instructions have been removed from the successor the scheduler then attempts to move the branch instruction(s) from the successor block into the C_block's penultimate long instruction word.

If a successor block can only be entered from the C_block, moving instructions across the block boundary will have no effect on the program. However if a successor block has more than one predecessor, compensation code must be introduced to preserve the correctness of the program. If the C_block's branch target block has more than one predecessor, a new block, containing the instructions which have been removed from the branch target block, is introduced into those paths which pass through the branch target block, but not through the C_block (see Figure 2). If the C_block's sequential successor can be reached from other blocks, then no conditional compaction is attempted, since a new branch instruction would be needed in the C_block to branch over the necessary compensation code.

If during the compaction process the C_block inherits a new successor block, the C_block is returned to the head of the compaction list and the process is repeated. For example consider the statements

```
        WHILE (testarray[j] > temp) DO
             j := j - 1;
        END;
        testarray[i]  :=  testarray[j];
```

which are taken from a procedure with declaration

```
         Quicksort (VAR testarray : intarray; bot, top : INTEGER);
```

which quick sorts an array of type [1..10] OF INTEGER.

Translating the statements into unconditionally executed sequential code results in three basic blocks comprising a total of 15 short instructions where, for simplicity, the code for bounds checking has been removed. Locally compacting each block gives:

Assume: R14, R19 and R13 contain j, temp and i respectively. P1 contains the address of the base of the actual parameter corresponding to testarray minus a constant component resulting from the non zero lower array bound[26].

```
WhileTst:   ASL  R21, R14, #2
            ADD  R22, P1, R21
            LD   R23, 0(R22)
            GTS  B7, R23, R19
            BF  B7, EndWhile
            NOP                                                 Block 1
---------------------------------------------------------------------------
            BRA  WhileTst        SUB  R14, R14, #1
            NOP                                                 Block 2
---------------------------------------------------------------------------
EndWhile    ASL  R24, R13, #2    ASL  R26, R14, #2
            ADD  R25, P1, R24    ADD  R27, P1, R26
            LD   R28, 0(R27)
            ST   0(R25),  R28                                   Block 3
---------------------------------------------------------------------------
```

Assuming that block 1 has reached the head of the compaction list; the scheduler selects block 1 as the C_block. Block 1 branches forwards, so its sequential successor block, block 2, is considered before its branch target block. Block 2 can only be entered from block 1 so the scheduler computes the CASet which is {T B7 SUB R14, R14, #1}. This instruction is scheduled in parallel with BF B7, EndWhile and the SUB instruction is removed from block 2. The conditionally executed instruction T B7 BRA WhileTst is then moved into the C_block's penultimate long instruction word, and block 2 is eliminated. The branch from block 1 to block 3 is then redundant and is removed. The label is removed from block 3, as it has no other predecessors, and the instruction T B7 BRA WhileTst is replaced with the equivalent, but unconditionally executed, instruction BT B7 WhileTst, giving:

```
WhileTst:   ASL  R21, R14, #2
            ADD  R22, P1, R21
            LD   R23, 0(R22)
            GTS  B7, R23, R19
            BT   B7, WhileTst        T B7  SUB  R14, R14, #1
            NOP                                                   Block 1
-------------------------------------------------------------------------
            ASL  R24, R13, #2        ASL  R26, R14, #2
            ADD  R25, P1, R24        ADD  R27, P1, R26
            LD   R28, 0(R27)
            ST   0(R25), R28                                      Block 3
-------------------------------------------------------------------------
```

Block 1 now has a new sequential successor block and a new branch target block, so it is returned to the compaction list, where it is reselected as the C_block and the compaction process is repeated. Block 1 branches backwards (to itself) so the scheduler attempts to compact instructions from the second iteration of the block which are conditionally executed on T B7. The CASet contains the first three instructions from block 1, but only the first instruction is scheduled, in the NOP slot, due to the definition-use dependency with respect to R14. Block 1 can also be entered from block 0 (not shown in the example); so a new block is created containing the ASL instruction and block 0 is returned to the compaction list. Finally the scheduler compacts instructions from block 1's sequential successor block, block 3, resulting in the following conditionally compacted code:

```
WhileTst:      ASL  R21, R14, #2                                                                    New Block

-------------------------------------------------------------------------------------------------------------

WhileTst+1     ADD  R22, P1, R21

               LD   R23, 0(R22)

               GTS  B7, R23, R19

               BT B7   WhileTst+1   T B7  SUB  R14, R14, #1   F B7  ASL  R24, R13, #2   F B7  ASL  R26, R14,#2

               NOP                  T B7  ASL  R21, R14, #2   F B7  ADD  R25, P1, R24   F B7  ADD  R27, P1, R26

                                                                                              Block1

-------------------------------------------------------------------------------------------------------------

               LD   R28, 0(R27)

               ST   0(R25), R28                                                               Block 3

-------------------------------------------------------------------------------------------------------------
```

The while loop has been reduced from eight long instructions in the locally compacted code to six long instructions on the first iteration of the loop, and five long instructions on the second and subsequent iterations of the loop. Furthermore the new block is subsequently removed when the compaction process continues and the ASL instruction is scheduled in block 0. Hence there is a saving of 4 cycles over the sequential code, and 3 cycles over the locally compacted code, each time the loop is executed.

EVALUATION

The combined effectiveness of the HARP architecture and the scheduling techniques was evaluated by comparing the performance of sequential and parallel compilations of eight benchmark programs running on simulations of the machine model. This paper compares program measurements obtained using the sequential simulation to those obtained using a simulation capable of executing 4 ALU, one Boolean, two memory reference and two branch instruction in parallel. This simulation was envisaged as providing sufficient functional units for the amount of parallelism detected by the compiler and subsequently proved to be closest to the iHARP implementation in terms of functionality. Although the simulation allows two memory reference instructions to be executed in parallel, the compiler only schedules two memory reference instructions in parallel if they are executed on mutually exclusive conditions. This restriction reflects the resource limitations of the iHARP processor which provides a single data memory port.

Table 3 lists the set of benchmark programs and shows the static instruction count and number of useful instructions executed for the sequential code (i.e. the dynamic instruction count excluding NOPs). The benchmarks comprise a set of short, general-purpose, integer computations. The first five programs were taken from the Stanford Small Programs Benchmark set[27], and adapted slightly to conform to the compilers source language (a subset of Modula-2). Three other well known programs were chosen to complete the benchmark set. Table 4 shows that an average instruction execution rate of 0.86 instructions per cycle is achieved for the sequential code produced by the HRC compiler, which makes no attempt to fill the branch delay slots. This rate is increased to 0.93 instructions per cycle assuming that a simple delayed branch scheme could be expected to fill 70% of the branch delay slots[28]. This result is within sight of the goal of single cycle execution for sequential code which has been optimised to remove pipeline hazards.

Local compaction results in execution rates ranging from 1.35 to 2.15 sequential instructions per cycle. Table 5 shows the performance of the locally compacted code for each of the benchmark programs. The four programs which perform best under local compaction all spend a significant proportion of their execution time in loops which contain large basic blocks which exhibit high degrees of compaction. The other four programs have shorter basic blocks which limit the potential for parallelism. The performance of Quick and Fib is further restricted by a high percentage of memory reference instructions. Since two unconditionally executed memory reference instructions can not be scheduled in parallel the second memory reference pipeline is never used during local compaction, and a block which contains m memory reference instructions must locally compact to at least m long instruction words. An average execution rate of 1.68 sequential instructions per cycle is achieved. This corresponds to speedups of 1.94 and 1.81 over the sequential code in which 0% and 70% of the branch delay slots have been filled. These figures are in line with Jouppi's findings[22] that parallelism within basic blocks is limited to a factor of 2 for the Stanford benchmark programs.

Finally Table 6 shows the performance of the conditionally compacted code for each of the benchmark programs. Conditional compaction results in execution rates ranging from 1.44 to 2.71 sequential instructions per cycle. In general the programs which perform best under local compaction also perform best under conditional compaction. The notable exceptions are BinSearch and Sieve. These programs achieve the biggest speedups when conditional compaction is applied to the locally compacted code. This is due to the low proportion of subroutine call and memory reference instructions executed by the programs. Blocks which end in a procedure call or return cannot be conditionally compacted; furthermore the second memory reference pipeline is only used if two mutually exclusive memory reference instructions can be scheduled in parallel (that is if a memory reference instruction from a block's branch target block can be scheduled in parallel with a memory reference instruction from its sequential successor). Thus if a block contains m memory reference instructions not only must it locally compact to at least m long instruction words but conditionally executed memory reference instructions from its successor blocks cannot be scheduled in these m words. Conditional compaction achieves an average execution rate of 2.18 sequential instructions per cycle. This corresponds to speedups of 2.53 and 2.35 over the sequential code in which 0% and 70% of the branch delay slots have been filled and a speedup of 1.31 over the locally compacted code. These results are encouraging and indicate that conditional compaction can be used to achieve sustained execution rates in excess of two sequential instructions per cycle for the HARP architecture.

CONCLUSIONS

This paper introduces local and conditional compaction; two compile-time scheduling techniques for HARP a new long instruction word architecture. These techniques have been implemented in a compiler which produces code for simulations of the HARP machine model. This model describes a class of long instruction word architectures with a variable number of instruction pipelines and forms the basis for the design of the iHARP VLIW processor which is currently under development.

Unlike many multiple-instruction-issue machines which are targeted at scientific applications, HARP is targeted at general-purpose computations. General purpose code is characterised by a high proportion of dynamic branches which can severely limit the potential for parallelism, even when the scope of the scheduler is extended beyond basic blocks. Hence the model provides several distinctive features, notably a compact four stage pipeline, an ORed addressing mechanism, unrestricted register bypassing and a two instruction branch architecture, which are specifically designed to reduce the number of delay slots which will effectively waste the available parallelism.

The local compaction algorithm uses list scheduling to schedule the unconditionally executed short instructions within basic blocks. Conditional compaction uses a modified version of the list scheduling algorithm to move instructions from a block's sequential and branch destination blocks in to the locally compacted schedule for the block. Instructions which are moved across a conditional branch are conditionally executed on the value of the Boolean variable which results in entry to their native block. Conditional execution removes the need for global data flow analysis, or a branch prediction scheme, and makes global scheduling relatively straightforward.

These techniques have been evaluated in conjunction with a simulation of the HARP model capable of executing a maximum of nine RISC type instructions in parallel. The experiments show that local compaction achieves an average speedup of 1.94 over sequential code, which is increased by a further 59% to a factor of 2.53 when the conditional compaction technique is used to schedule instructions across basic block boundaries. This represents an average instruction execution rate of 2.18 sequential instructions per cycle which is in line with the project objective of a sustained execution rate in excess of two instructions per cycle. However the performance of the scheduling algorithms is still limited by the absence of any memory reference disambiguation and the restriction to a single data memory port. Consequently work is now underway to to assess the effect of introducing memory reference disambiguation into the compiler, while allowing the parallel execution of two, or possibly more, memory reference instructions.

ACKNOWLEDGEMENTS

The authors acknowledge the support of the other members of the HARP team: Paul Findlay, Brian Johnson, Simon Trainis, Dave McHale and Liang Wang. They would also like to thank Dr S. L. Stott, Professor L. C. W. Dixon, Dr P. Kaye and J. A. Davis for their support throughout the project.

The HARP project is supported by SERC Research Grant GR/F88018. Part of this work was also supported by a SERC studentship.

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 and 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 **Groves, R D and Oehler, R** 'RISC System/6000 processor architecture' *Microprocessors and Microsystems* Vol 14 No 6 (July/August 1990) pp 357-366

4 **Lee, R L, Kwok, A Y and Briggs, F A** 'The Floating Point Performance of a Superscalar SPARC Processor' *Proc. 4th Int. Conf. Architectural Support for Programming Languages and Operating Systems* (April 1991) pp 28-37

5 **Hwu, W W and Patt, Y N** 'HPSm, A High Performance Restricted Data Flow Architecture Having Minimal Functionality' *Proc. 13th Ann. Int. Symp. Computer Architecture* (June 1986) pp 330-336

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

7 **Rau, B R, Yen, Y W and Towle, R A** 'The Cydra 5 Departmental Supercomputer' *IEEE Computer* (January 1989) pp 12-35

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

9 **Cohn, R, Gross, T, Lam, M and Tseng, P S** 'Architecture and Compiler Tradeoffs for a Long Instruction Word Microprocessor' *ASPLOS III* Boston (April 1989) pp 2-14

10 **Smith, J E** 'Dynamic Instruction Scheduling and the Astronautics ZS-1' *IEEE Computer* Vol 22 No 7 (July 1989) pp 21-35

11 **Murakami, K, Irie, N, Kuga, M and Tomita, S** 'SIMP: A Novel High-Speed Single-Processor Architecture' *16th Ann. Int. Symp. Computer Architecture* (May 1988) pp 78-85

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

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

14 **Ebcioglu, K and Nakatani, T** 'A New Compilation Technique for Parallelizing Loops with Unpredictable Branches on a VLIW Architecture' *Languages and Compilers for Parallel Computing* D. Gelernter et al. (eds.), Research Monographs in Parallel and Distributed Computing, MIT Press (1989)

15 **Smith, M D, Lam, M and Horowitz, M A** 'Boosting Beyond Static Scheduling in a Superscalar Processor' *Proc. 17th Ann. Int. Symp. Computer Architecture* (May 1990) pp 344-353

16 **Weiss, S and Smith, J E** 'A Study of Scalar Compilation Techniques for Pipelined Supercomputers' *Proc. 2nd Int. Conf. Architectural Support for Programming Languages and Operating Systems* (October 1987) pp 105-109

17 **Lam, M** 'Software Pipelining: An Effective Scheduling Technique for VLIW Machines' *Proc. ACM SIGPLAN '88 Conf. Programming Language Design and Implementation* (June 1988) pp 318-327

18 **Steven, G B and Gray, S M** 'Specification of a Machine Model for the HARP Architecture and Instruction Set - Version 3' *Computer Science Technical Report No 117* Hatfield Polytechnic, UK (January 1991)

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

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

21 **Gray, S M** 'Code Generation for a Long Instruction Word Architecture' *PhD Thesis* Hatfield Polytechnic, UK (1991)

22 **Jouppi, N P and Wall, D W** 'Available Instruction Level Parallelism for Superscalar and Superpipelined Machines' *Proc. 3rd Int. Conf. Architectural Support for Programming Languages and Operating Systems* (April 1989) pp 272-282

23 **Green, G J** 'A Simulation of a HARP architecture in ELLA' *Computer Science Technical Report No 104* Hatfield Polytechnic, UK (July 1990)

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

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

26 **Gray, S M** 'The Implementation of Arrays in the HARP Research Compiler' *Computer Science Technical Report No 116* Hatfield Polytechnic, UK (December 1990)

27 **Weicker, R P** 'An Overview of Common Benchmarks' *IEEE COMPUTER* (December 1990) pp 65-75

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

Table 1. HARP Instruction Types

Instruction Type	Example	Semantics
Computational	ADD R22, R21, R20	R22 := R21 + R20
Relational	LT B2, R21, R20	B2 := R21 < R20
Boolean	AND B4, B3, B2	B4 := B3 AND B2
Branch	BT B3, Lab1	If B3 = TRUE branch to Lab1
Memory Reference	ST (R20,R21), R22	address (R20, R21) := R22

Table 2. Partial ordering of short instructions maintained to preserve data integrity.

Dependency	Partial Ordering
s_i defines a register or memory location used by s_j	s_i must execute before s_j
s_i and s_j define the same register or memory location	s_i must execute before s_j
s_j defines a memory location used by s_i	s_i must execute before s_j
s_j defines a register used by s_i	s_i must execute before or in parallel with s_j.

Table 3. The Benchmark Programs

Program	Sequential Code		Description
	Static Instrs.	Dynamic Instrs.	
Bubble	216	2535	Bubble sort
Quick	279	1708	Recursive quick sort
Perm	176	27129	Recursive computation of permutations
Queens	232	48320	Recursive solution of the 8 queens chess problem
Intmm	206	6584	Integer matrix multiplication
BinSearch	90	220	Iterative Binary Search
Fib	73	1046	Recursive computation of Fibonacci numbers
Sieve	78	8217	Sieve of Eratosthenes

Table 4. Performance of Sequential Code

Program		Sequential Code	
	Cycles	Instrs/Cycle	Estimated Instrs/Cycle using delayed branch scheme
Bubble	2939	0.86	0.92
Quick	2048	0.83	0.90
Perm	30238	0.90	0.96
Queens	51862	0.93	0.98
Intmm	7991	0.82	0.86
BinSearch	257	0.86	0.95
Fib	1256	0.83	0.92
Sieve	9679	0.85	0.95
Average		0.86	0.93

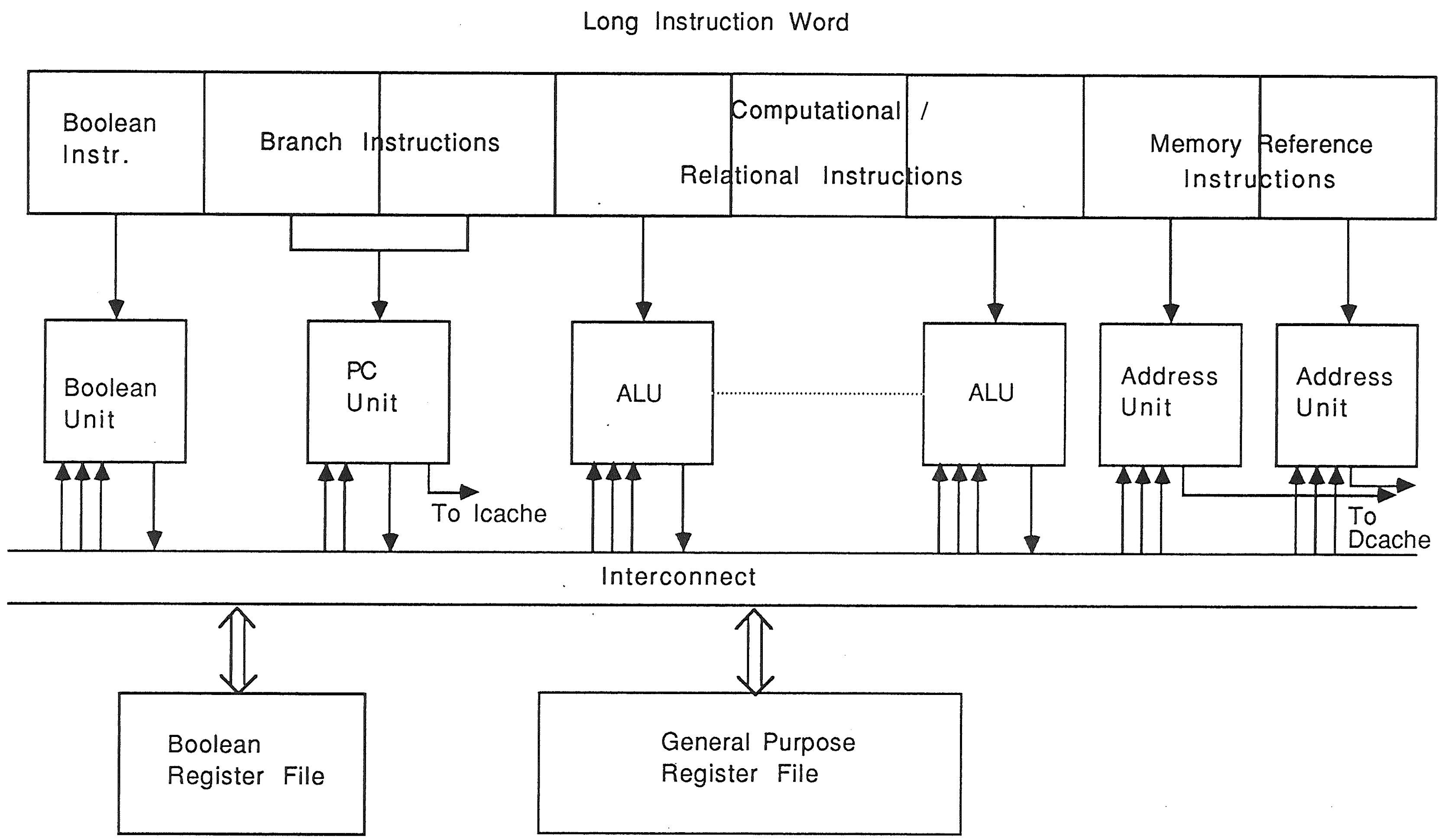
Table 5. Performance of Locally Compacted Code

Program	Locally Compacted Code			
	Cycles	Sequential Instrs/Cycle	Speedup over sequential code	
			0% branch delay slots filled	70% branch delay slots filled
Bubble	1519	1.67	1.94	1.82
Quick	1268	1.35	1.63	1.50
Perm	14760	1.84	2.04	1.92
Queens	22430	2.15	2.31	2.19
Intmm	3292	2.00	2.44	2.33
BinSearch	139	1.58	1.84	1.66
Fib	756	1.38	1.66	1.50
Sieve	5706	1.44	1.69	1.52
Average		1.68	1.94	1.81

Table 6. Performance of Conditionally Compacted Code

Program	Conditionally Compacted Code				
	Cycles	Sequential Instrs/Cycle	Speedup over sequential code		Speedup over locally compacted code
			0% branch delay slots filled	70% branch delay slots filled	
Bubble	1212	2.09	2.42	2.27	1.25
Quick	1030	1.66	1.99	1.84	1.23
Perm	14180	1.91	2.13	1.99	1.04
Queens	18017	2.68	2.88	2.73	1.24
Intmm	2429	2.71	3.29	3.15	1.36
BinSearch	86	2.56	2.99	2.69	1.62
Fib	728	1.44	1.73	1.57	1.04
Sieve	3415	2.41	2.83	2.54	1.67
Average		2.18	2.53	2.35	1.31

LIST OF FIGURE CAPTIONS



Long Instruction Word
Boolean Instr.
Branch Instructions
Computational / Relational Instructions
Memory Reference Instructions
Boolean Unit
PC Unit
To Icache
ALU
ALU
Address Unit
Address Unit
To Dcache
Interconnect
Boolean Register File
General Purpose Register File