DIVISION OF COMPUTER SCIENCE

An Evaluation of the Architectural Features of the iHARP Processor

Fleur L. Steven
Gordon B. Steven
Liang Wang

Technical Report No.170

December 1993

An Evaluation of the Architectural Features of the iHARP Processor

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

ABSTRACT

RISC processors have approached an execution rate of one instruction per cycle by using pipelining to speed up execution. However, to achieve an execution rate of more than one instruction per cycle requires multiple instruction issue (MII) processors which employ multiple pipelines. This paper evaluates the important architectural features of iHARP, the University of Hertfordshire's VLIW processor. Using a resource limited scheduler (RLS), the work shows that the inclusion of various architectural features, for example, conditional instruction execution or the increase in the number of data cache memory ports can improve the performance of a MII processor. A review of the work undertaken by a number of groups in the areas of potential instruction level parallelism and static scheduling show that a great amount of fine-grained parallelism is theoretically available. However, for a processor with four pipelines, our work achieves an instruction execution rate approaching two instructions per cycle.

1. INTRODUCTION

The 1980s was the decade where RISC ideas were very much in the forefront. During this decade the execution of one instruction per cycle was approached by using pipelining to speed up execution [Hennessy 90]. However, if the one instruction per cycle barrier was to be broken a different architectural approach would be required. Thus, the early 90s saw the emergence of architectures which issued more than one instruction in the same cycle. These architectures are referred to as multiple instruction issue architectures. iHARP is one such processor.

The computer architecture group at the University of Hertfordshire was formed in 1986 to research and develop novel architectures. The long term aim of the group is to develop VLIW and superscalar architectures to exploit fine grained parallelism with the objective of achieving an order of magnitude speed up over RISC. In support of this aim the group has developed iHARP a VLIW (Very Long Instruction Word) processor. To date rates in excess of two instructions per cycle have been achieved [Wang 93].

iHARP is a multiple instruction issue (MII) processor which fetches a 128 bit long instruction word from an instruction cache each cycle. Each long instruction comprises four, 32 bit short RISC primitives. These are dispatched to four integer pipelines for parallel execution. iHARP is a VLIW processor which therefore relies on compiler/scheduler software to detect groups of instructions which can be executed in parallel and to place these groups into a long instruction word at compile time. This approach contrasts with the superscalar [Johnson 91] approach where it is left to the hardware to detect and issue instructions which can be executed in parallel at run time.

The iHARP processor is a single chip, aimed at supporting non-numeric applications. Many multiple instruction issue processor projects target their processors at numeric applications which have more potential parallelism to exploit. Non-numeric code tends to be control intensive thereby restricting any parallelism potential. Research at Stanford University and DEC has shown that the average degree of exploitable instruction-level parallelism in typical programs is two and rarely exceeds three to four within a basic block [Wall 91]. Therefore, aiming a processor at non-numeric code and achieving an execution rate greatly in excess of one instruction per cycle requires not only a finely tuned processor architecture with multiple parallel pipelines but also software which can exploit the parallelism offered by a program. Therefore, an efficient instruction scheduler/compiler is an essential part of a VLIW processor project. To this end, a scheduler has been developed[Wang 93].

The same scheduling ideas developed for VLIW processors can also be applied to a superscalar processor and the computer architecture group is also developing a superscalar processor (HSP)[Steven 93][Collins 93] using the ideas produced during the HARP project.

This paper evaluates eight of the most important architectural features of the iHARP processor. The first is the instruction issue rate which has a direct effect on the number of instructions which can be executed each cycle. The second is code expansion which is one of the major disadvantages of VLIW architectures. The third is the number of cache memory ports as more ports relieve the data bottleneck. The fourth feature is conditional instruction execution. Very little evaluation of this feature has been undertaken. The fifth feature examined is ORed indexing which is an addressing mechanism allowing the removal of one of the stages of a five-stage pipeline and also the removal of any load delay. The sixth feature is the number of register writeback ports as it is useful to be able to return more than one result to the register file each cycle. The penultimate feature examined is the number of branch units as iHARP allows two branches to be executed in parallel. Finally, the effect of combining shifts with arithmetic operations (combined instructions) is examined.

The remainder of this paper is organised as follows: Section 2 discusses the problems associated with data dependencies. Section 3 reviews other research being undertaken in the area of static instruction scheduling. Section 4 briefly describes the compiler/scheduler software and the HARP models used in this research. Section 5 considers the architectural features of iHARP examined in this paper and presents the results obtained. Section 6 offers some concluding remarks.

2. THE REMOVAL OF DATA DEPENDENCIES

Instruction scheduling attempts to place instructions in parallel by moving them up to as early a point in the code sequence as possible thereby reducing the average number of cycles taken per instruction. However, there are inherent limitations to the amount of parallelism that can be obtained owing to data dependencies between pairs of instructions.

In general three data dependencies can be identified. The first is a true data dependency, or read after write (RAW). A RAW data dependency occurs when instruction i + 1 wants to read data modified by instruction i. The second is an anti-dependency, or write after read (WAR). A WAR data dependency occurs when the write operation of instruction i + 1 destroys a value read by i. Finally, the third is an output dependency, or write after write (WAW). A WAW data dependency occurs when instructions i and i + 1 write to the same destination. WAR and WAW dependencies are not true data dependencies, rather they are name conflicts which can be eliminated by renaming registers during instruction scheduling. The instruction sequence below shows how data dependencies can be removed by using register renaming:

```
I1  ADD R5, R6, R7
I2  ADD R5, R5, R6
I3  ADD R6, R7, #256
```

Instruction I2 has a true data dependence on I1 and instruction I3 has an anti-dependency on I2. However, if the output of I3 is renamed to register R20 and the result is copied to register R6 later, instruction I3 can be moved ahead of both I1 and I2 as shown below:

```
I3  ADD R20, R7, #256
I1  ADD R5, R6, R7
I2  ADD R5, R5, R6
    ..
    ..
    ..
    MOV R6, R20
```

Note that the copy instruction in the example above need not introduce further data dependencies as any later instructions requiring R6 as an input can equally well use R20.

As another example consider the following code sequence:

```
I1  ADD R5, R6, R7
I2  ADD R6, R5, R6
I3  ADD R5, R7, #256
```

Instruction I2 has a true data dependency on I1 and instruction I3 has an output dependency on I1 as well as an anti-dependency on I2. If the output of I3 is renamed to register R20 and the result copied to register R5 later, instruction I3 can be moved ahead of both I1 and I2. Again subsequent instructions can use R20 instead of R5 to avoid the introduction of additional data dependencies as shown below:

```
I3  ADD R20, R7, #256
I1  ADD R5, R6, R7
I2  ADD R6, R5, R6
    . .
    . .
    . .
    MOV R5, R20
```

Register renaming is also useful for moving code across edges between basic blocks[1]. For example consider the following piece of code:

```
NE B6, R1, R2        ; set B6 if R1 <> R2
BT B6, Label         ; if B6 is true branch to Label
NOP

ADD R6, R7, R8                          Label: ...
```

In iHARP conventional condition codes are replaced by eight one-bit boolean registers which are explicitly set by relational instructions. For example, in the above code the instruction NE B6, R1, R2 sets boolean register B6 to TRUE if the value of register R1 is not equal to the value of register R2. The branch instruction then uses register B6 to decide whether to jump to Label or not. Thus, if B6 is TRUE execution will continue at Label otherwise execution will continue sequentially.

The ADD instruction could be moved up after the relational instruction and executed speculatively. Speculative execution allows an instruction to be executed before it is known whether the path containing that instruction is actually taken. The code sequence with the ADD instruction moved up is shown below:

```
NE B6, R1, R2
ADD R6, R7, R8
BT B6, Label
NOP                                     Label: ...
```

However, if R6 is live on the Label path and the ADD is moved up before the branch it will have executed and changed the R6 value incorrectly in the case where the branch is taken.

One solution to the above problem is to use conditional or guarded execution. iHARP uses the boolean registers which are set explicitly by relational instructions to control the execution of subsequent instructions. For example, the instruction TB1 ADD R1, R2, R3 will only add R2 to R3 and place the result in R1 if boolean register B1 holds the value TRUE.

Applying conditional execution to the example code it becomes:

```
      NE B6, R1, R2
FB6   ADD R6, R7, R8; If B6 is false R6 := R7 + R8
      BT B6, Label
      NOP                               Label: ...
```

If register R6 is now live on the Label path the value will be the correct one because the ADD instruction is prevented from executing by the guard register B6. Therefore, the ADD instruction is no longer speculatively executed.

The ADD instruction can not be moved up any further because there will be no guard available to prevent it from executing. However, register renaming can be used to allow further code motion. The example using register renaming is shown below:

[1] A sequence of code with no jumps in except at the beginning and no jumps out except at the end.

```
NE B6, R1, R6
ADD R16, R7, R8
BT B6, Label

MOV R6, R16                   Label: ...
```

Register R6 is replaced by register R16 in this example. If the branch is taken and R6 is live on the label path, R6 will hold the correct value since it has not been changed at this stage. A MOV instruction is used to copy the contents of R16 into R6 if the branch is not taken. The added advantage of register renaming is that the ADD instruction can now be moved up further, before the relational instruction if necessary. Thus, R16 could be used anywhere R6 is used thereby preventing the inhibition of code motion. However, conditional execution does have three advantages when compared with register renaming. The first advantage is that a copy will not be required thereby conserving resources. The second advantage is that live variable analysis is not required. The third advantage is that it reduces the number of registers used.

The above code example can also be used to illustrate a problem associated with speculative execution. During scheduling an instruction may be moved up into an available slot which results in it accessing an invalid memory location even if it is not subsequently required, that is, if it is on the wrong branch path. The exception generated by the rescheduled instruction will be spurious since the instruction would not normally be executed. Therefore, two forms of each instruction must exist; a speculative form and a normal form. The normal form generates an immediate interrupt. The speculative form will not generate an immediate interrupt but will result in an interrupt if the speculative result is used later in a non-speculative instruction. Speculative instructions are implemented in the following manner: An exception in a speculative instruction generates a polluted value in the result register. As long as the result is only used by a speculative instruction further polluted values are generated. If a polluted result is subsequently used by a non-speculative instruction then an interrupt will result. This allows both loads and ALU operations to be executed speculatively. The same idea can be applied to relational operations if a flag is added to each boolean register to indicate when a polluted value is held.

For example, in the code sequence below the load instruction can be moved up just as the ADD instruction was in the earlier example:

```
NE B6, R1, R2
BT B6, Label
NOP

LD R6, (R0, R5)               Label: ...
```

Using conditional execution the code becomes:

```
    NE B6, R1, R2
FB6 LD R6, (R0, R5)
    BT B6, Label
    NOP                       Label: ...
```

Again, the instruction guarded by B6 will not be executed if the branch is taken.

However, if the register R6 is now renamed to R16 as shown below

```
NE B6, R1, R2
LD R16, (R0, R5)
BT B6, Label
NOP                           Label: ...
```

there will be a problem if the branch is taken because the memory location at (R0, R5) will still be accessed possibly causing an exception if it is now an invalid memory address. Therefore, support will be required for speculative execution if register renaming rather than conditional execution is the method used.

iHARP's conditional execution can also be employed when multiple guard conditions are required. The following example involves two branch instructions scheduled in parallel.

```
LIW  i      NOP;  BT B1, L1;  OR B3, B1, B2;  FB1 BT B2, L2
LIW  i + 1  FB3 ST (R0, R8), R6;  NOP;  NOP;  NOP
```

The scheduler has generated a boolean instruction, 'OR B3, B1, B2', to compute an appropriate guard for the store instruction which has been scheduled in the final long instruction word.

3. A REVIEW OF WORK UNDERTAKEN IN THE AREA OF STATIC INSTRUCTION SCHEDULING

The investigation of the limits of potential instruction level parallelism and static instruction scheduling is being undertaken by many groups. This section reviews the methods and results produced by these groups.

3.1 Potential Instruction Level Parallelism

Wall [Wall 91] investigated the available parallelism in 17 test programs including the SPEC benchmarks. These programs were executed to produce a trace of instructions executed plus data addresses referenced and the results of branches and jumps.

It was further assumed that any cycle could contain up to 64 instructions in parallel. There were no further limits on replicated functional units or ports to registers or memory. All 64 instructions could be multiplies or loads. Also every operation had a latency of one cycle. Further, it was assumed that cache hit rates were 100%. The size of the instruction window was 2048 instructions by default but could be varied between four and 2048 instructions. Windows could be managed either discretely or continuously. Discrete management involves an entire window being fetched then scheduled before a fresh window is used. Continuous management involves a new instruction entering the window one at a time and old instructions leaving the window whenever the number of instructions reaches the window size. Continuous window management was used to obtain the results in this investigation.

Wall assumed that there were three kinds of register renaming: perfect register renaming which allowed an infinite number of registers; finite register renaming which allocated a set of 256 registers using a LRU algorithm; and no register renaming at all.

Various types of branch prediction were also provided. These were perfect branch prediction, two-bit branch prediction both infinite and finite, static prediction and an option that no branch prediction would be provided. Perfect prediction assumed all branches were correctly predicted. Infinite two-bit prediction meant that two different branches never had the same table entry because an infinite sized branch prediction table was used whilst finite two-bit prediction used a table limited to 2048 entries. Static prediction predicted that a branch would always go the way it goes most frequently. The same schemes were available for jump prediction.

A series of five models was defined from "stupid" which provided no branch predicition, register renaming or alias analysis to "perfect" which provided perfect branch prediction, register renaming and alias analysis.

The results showed that infinite, finite and static branch prediction all did equally well. The "stupid" model with no branch prediction, register renaming or alias analysis rarely achieved a parallelism above two. The "fair" model with infinite branch prediction and finite register renaming achieved a parallelism between two and four. The "great" model with infinite branch prediction and infinite register renaming achieved a parallelism of approximately 8.

The conclusions arrived at by Wall were that good branch prediction seemed necessary for extraction of more than modest amounts of instruction level parallelism. Register renaming was also found to increase the amount of parallelism obtained. The average amount of parallelism found for a hardware-style model with branch and jump prediction using infinite tables, 256 FPU and 256 CPU registers and windows of 64 instructions maintained continuously was approximately 7 with the median approximately 5.

Butler, Yeh and Yale Patt [Butler et al 91] also investigated potential instruction level parallelism using nine integer and floating point programs from the SPEC suite. These were run on an instruction level simulator for the MC88100 superscalar with different machine configurations.

Each machine was specified by several important factors. These factors were as follows: the set of functional units provided; the branch prediction mechanism; the issue rate; the size of the instruction; the characteristics of the load/store pipelines; and the instruction prefetch buffer configuration. The prefetch buffer configuration comprised the size of the buffer, refill characteristics and window size (the total number of instructions that can exist in the machine at any one time). The window size is measured in instruction packets where the size of an instruction packet is the group of instructions that are issued in a single cycle. A flag indicated whether instructions could be executed in-order or out-of-issue order for each functional unit. The machine configurations used ranged from machines, named UDF and RDF.I8, with unlimited numbers of functional units each of which was capable of executing all instruction classes to one with only four functional units, each functional unit only executing certain classes of instructions. All the machine configurations use branch prediction. There are two classes of branch prediction in this scheme and they are termed synthetic and real. With synthetic branch prediction, the branch prediction accuracy is specified in the machine configuration file. As branches are encountered in the dynamic instruction stream, a random number is generated to determine whether the branch is predicted correctly or not. In Real branch prediction an actual prediction is made and if it fails the pipeline stalls until the branch is resolved.

The simulator for the MC88100 reads in object code and simulates execution producing an instruction trace. The RDF simulator reads in a configuration file describing the machine to be simulated and then begins processing the dynamic instruction stream produced by the MC88100 simulator. The simulator performs data dependency analysis and scheduling and gathers execution rate statistics.

The highest performance from RDF.I8 which restricted window size to a maximum of 16 packets and the fetch rate to 8 instructions per cycle was 8 instructions per cycle. This model assumed branch prediction was always correct and provided an upper bound for machines approximating this model.

The UDF machine did not restrict window size or fetch rate and provided an absolute upper bound for a particular benchmark. The parallelism achieved ranged from 17 to 1165 instructions per cycle.

Each of the other machine configurations combined realistic window size, issue rate, functional units and one of the following branch prediction schemes: 100% accurate; 85% accurate (synthetic); and real branch prediction.

Machines with four functional units, two memory units and 100% branch prediction produced a result of 2.7-2.9 instructions per cycle for integer benchmarks and 1.5-2.8 for floating point. When the branch prediction was 85% the figures were lower at 1.9-2.2 for integer and 1.4-2.7 for floating point.

At six functional units and three memory units and 100% branch prediction the figures were higher at 3.2-3.6 for integer and 1.8-5.7 for floating point.

Several machines used 8 functional units with three memory units. With 85% branch prediction the figures were 2.2-2.9 for integer and 1.6-5.0 floating point. Using real branch prediction gave slightly higher figures of 2.4-3.4 for integer and 1.9-5.9 for floating point.

To summarise, Butler achieved results of between approximately 2 and 5.8 instructions per cycle. The machine with unlimited resources gave results of between 17 and 1165 instructions per cycle. They concluded that parallelism in excess of 5 instructions per cycle was certainly possible.

Lam [Lam 92] investigated the constraints imposed by control flow on parallelism using six SPEC benchmarks and four others. Three techniques were used to relax these constraints: control dependence analysis; executing multiple flows of control simultaneously; and speculative execution.

Seven abstract machines were specified. The BASE machine used none of the three techniques and acted as a baseline for comparison. The ORACLE machine used perfect branch prediction and no constraints due to control flow and acted as an upper bound of parallelism. The other five used a combination of control dependence analysis (CD), multiple flows of control (MF) and speculative execution (SP). For example, the CD-MF machine allowed branch instructions to execute in parallel and control dependence branches had to be resolved before an instruction could execute. All machines with SP only speculatively executed instructions on the most likely execution path.

The benchmarks were compiled for the MIPS R3000 processor by MIPS C and FORTRAN compilers with full optimisation. The traces were obtained using the MIPS Pixie tool and each program was simulated for up to 100 million instructions. All instructions had a latency of one cycle.

The results ranged from 2.14 for the BASE machine to 158.26 for the ORACLE machine. For the CD machine the branches must execute in order at one per cycle to reflect the inability to pursue multiple flows of control. It achieved 2.39 instructions per cycle. The CD-MF machine achieved 6.96 instructions per cycle.

Lam concluded that architectures lack support for control flow. Speculative execution is necessary to find sufficient parallelism. Lam also suggested that guarded execution on its own is inefficient for following multiple complex flows of control simultaneously and requires both guarded execution and speculative execution for higher performance.

There are three factors which can be concluded from studies such as those above. First, these and other theoretical studies are useful in predicting the upper bound on potential parallelism and therefore whether or not it is worth looking for more parallelism. Second, since the amount of resources assumed in such studies is unrealistic the parallelism obtained can not be directly compared with results from real machines where resources are by necessity limited. Third, the results of these studies have shown us that there is lot of parallelism theoretically available, in fact, more than current algorithms can successfully exploit.

3.2 Scheduling Techniques

List Scheduling [Landskov et al 80] is a local microcode compaction technique of polynomial complexity which uses non-backtracking heuristics to produce a near optimal instruction schedule and is also used to exploit parallelism in MII architectures. List Scheduling schedules a basic block consisting of an instruction set I = {i1, i2, ..., i*n* } into a long instruction word set W = {L1, L2, ..., Lp} for a specific MII architecture. In general, List Scheduling can produce a near optimum length schedule W for a given I.

List Scheduling assigns a priority to each instruction and schedules each in turn. Scheduling an instruction using List Scheduling depends upon three factors. First, if instruction i_k has a true data dependency on i_j then i_k must not be scheduled in W ahead of i_j. Second, if instruction i_j and i_k are independent, then the one with higher priority is scheduled first. Typically instructions which initiate a long chain of computations are given a higher priority than those which initiate a shorter chain of computations or none at all. Third, when an instruction is placed in a non empty L, resource constraints on L must be checked to prevent a false schedule.

The full benefit of List Scheduling is only realised when a group of basic blocks are combined to form a single large basic block. Trace Scheduling treats a trace of multiple basic blocks as a single unit for the purposes of List Scheduling. Trace Scheduling is described below.

Fisher was the leader of the ELI-512 VLIW project. He developed a technique termed Trace Scheduling [Fisher 81] which uses software branch prediction to combine basic blocks that are usually executed in sequence into larger blocks called traces. It was originally developed as a technique for scheduling and packing operations into horizontal microinstructions and has evolved into the principal technique used in VLIW architectures for numeric applications [Fisher 83][Colwell 87].

Traces are treated as single basic blocks and are scheduled using List Scheduling. Loops, especially the innermost loops, are unrolled many times to give large traces. The most frequently executed traces, in particular, innermost loops, are scheduled first. Once a trace has been compacted, it will not be affected by the later compaction of other traces. When List Scheduling a trace, program correctness is ensured by book keeping techniques which insert compensation code into the off-trace blocks. Trace scheduling is a better technique for numeric code than non-numeric code as it is easier to obtain long traces by unrolling loops. This process improves the performance of favoured traces at the expense of later traces and also leads to a rapid increase in code size because of loop unrolling and compensation code.

The results for code size expansion from the work on trace scheduling showed that code size increased by 30%-60% when using inter block trace selection, loop unrolling and procedure inlining. When several large (100K-300K) FORTRAN programs were used with unrolling and trace selection, code size was approximately 3 times larger than VAX object code (compiled with a VAX/VMS FORTRAN compiler) [Colwell 87].

Speedup varied depending upon the ELI machine being used and therefore the number of resources available. Work undertaken by Fisher, Ellis and others at Yale [Fisher 84] using an "ideal" machine with infinite resources and a "Simple" ELI machine with 8 clusters[2] and 4 functional units connected by a complete crossbar to multi-ported register bank showed that for a Fast Fourier Transform program the speedup was 47 for the "ideal" machine and 7.5 for the "simple" machine. Two other machines were described in the paper, a "realistic" ELI machine also incorporating 8 clusters and very close to the actual ELI being designed at that time, and a pipelined sequential machine resembling the CDC 6600. However, figures for these other two machines are not given in the paper.

Instruction boosting [Smith 92] was developed at Stanford using the MIPS R2000 and is a hardware mechanism to support speculative execution of instructions. In order for the boosting hardware to correctly maintain the program semantics the compiler must communicate its assumptions to the hardware. Thus, whenever the compiler moves an instruction above a control dependent branch, the compiler labels this instruction as a boosted instruction. This labelling encodes the control dependence information needed by the hardware so that the hardware can determine when the effects of the boosted instruction are no longer speculative. The labelling indicates which branch or branches the boosted instruction is control dependent upon and also indicates the predicted direction of each of these branches. A speculative instruction that is moved above *n* control dependent branches is referred to as an instruction that is boosted *n* levels.

[2] A cluster is a RISC with local registers, instruction memory, optional data memory, a few functional units implementing integer and floating point operations and a partial crossbar connecting these elements within it.

For example, an instruction "y := y + 1" in block B could be moved up to block A. As part of the processing a suffix is added to the destination operand as shown below:

y.BL := y + 1 ; "L" means left else "R" means right.

"BL" indicates to the hardware that "y" is committed if the branch in block A goes to the left. Similarly, x.BLL := x + 1 means that "x" is committed only if the next two branches all go to the left.

When an instruction is moved ahead of a branch at compile time, at run time its uncommitted result will be stored in shadow storage which consists of shadow register files and a shadow storage buffer. If later the branch outcome is committed, the value stored in shadow storage is transferred to the main register file or memory. If the branch outcome is not committed, the value stored in shadow storage is squashed.

Boosting avoids both extra renaming code and live variable analysis. However, it requires extra resources both in terms of a shadow register structure and extra instruction bits to define the level of boosting required.

The results of the work by Smith show that a single level of boosting achieves a 17% improvement over a base superscalar processor which is similar to models commercially designed today, that is, a small issue rate, typically two which is restricted in that not all pairs of instructions can be issued together, and a single data memory port. When boosting levels are increased to three and seven the performance improves by a further 2.3% and 3.5% respectively over the base processor.

Conditional compaction [Gray 91] is a global scheduling technique based on conditional instruction execution and is the first scheduling method developed at the University of Hertfordshire. Conditional compaction first uses List Scheduling to locally compact each basic block into long instruction words. A systematic attempt is then made to move instructions from each basic block into its predecessors. To preserve program semantics a boolean guard is added to any instruction moved.

Conditional compaction avoids the need for live variable analysis. However, it requires four extra instruction bits for defining the boolean register and boolean condition and a limited amount of parallelism is actually obtained owing to the limited amount of code motion possible using conditional compaction alone.

The results of the conditional compaction algorithm, using Stanford benchmarks and a GNU CC HARP compiler, show that for an issue rate of seven a speedup of 1.72 was obtained. This is significantly lower than the 1.96 obtained by the scheduler used in our evaluation for an issue rate of five using both register renaming and conditional execution.

Lam was a member of the WARP systolic array computer team at Carnegie Mellon and extended a well known technique termed Software Pipelining [Lam 88]. Software Pipelining is a method of overlapping operations from different loop iterations, without initially unrolling the loop, thereby, attempting to produce a minimum number of cycles between successive loop iterations. Software Pipelining is most successful for simple loops consisting of one basic block. Lam incorporates software pipelining into a global scheduling system called hierarchical reduction. In hierarchical reduction the innermost programming constructs are always scheduled first. Scheduled units are then incorporated into the next level of the scheduling hierarchy as a single complex node. Scheduling continues until the whole scheduling unit is reduced to a single node. Hierarchical scheduling always ensures that scheduling is performed in an optimum order.

For example, an if-then-else construct within an inner loop would be scheduled first. When software pipelining is applied to the loop, the if-then-else schedule is treated as a single node within the loop. The scheduled loop and its associated prologue and epilogue are then incorporated into the next level of scheduling. The new loop body is treated as a single node and code motion is allowed across the loop body, subject to appropriate scheduling restraints. In contrast, the epilogue and prologue are simply merged into the code surrounding the loop.

Nicolau was a member of the ELI-512 VLIW project at Yale and developed Percolation Scheduling [Nicolau 85]. Percolation Scheduling provides a set of core transformations which allow instructions, including branch instructions, to "percolate" or move from various parts of the program graph towards the beginning of the program graph. Initially each node in the program graph represents an individual instruction. However, as percolation proceeds the nodes are transformed into VLIW instructions.

The four core transformations are as follows:

- Deletion

If all the instructions in a node have been percolated out, the empty node can be deleted. The branch instructions in the predecessor nodes must be altered to point to the successor of the deleted node (Fig 3.1).

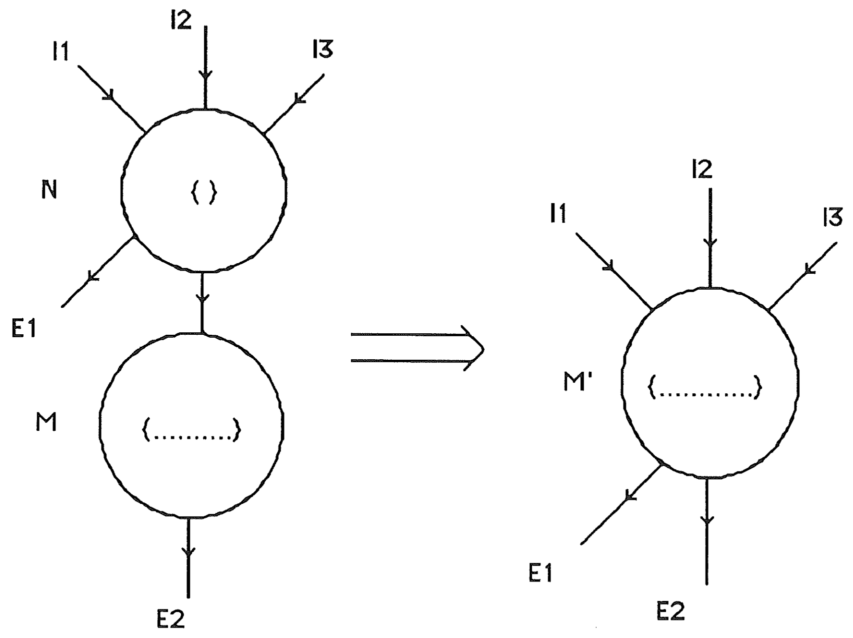


Fig 3.1 Deletion Transformation

- Move-op

An instruction is moved in parallel with its predecessors if there are no data dependencies. When an instruction is percolated beyond a convergence point, the node originally containing the instruction must be duplicated to preserve program correctness. Each copy will then be percolated up along its own path, possibly being split again, until a true data dependence prevents further percolation or the top of the graph is reached (Fig 3.2).

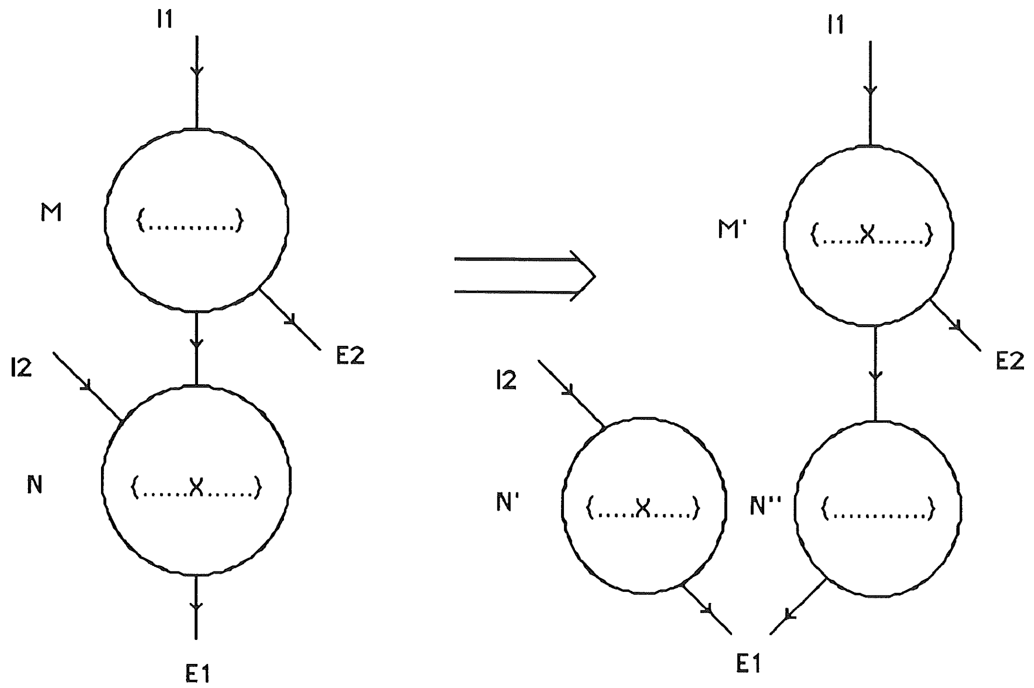


Fig 3.2 Move-op Transformation

- Move-cj

A conditional branch instruction is moved in parallel with its predecessors if there are no data dependencies. When the branch instruction is percolated beyond a convergence point, the node originally holding the branch instruction must be duplicated to preserve program correctness. The node originally containing the branch must be replicated and inserted in the two successor paths of the conditional branch (Fig 3.3).

$$\begin{tikzcd}
 & & M: \{.....\} \arrow[dll] \arrow[d] & \\
E3 & & N: \{....Cj....\} \arrow[dl] \arrow[dr] & \\
 & E1 & & E2
\end{tikzcd}$$

Moving Cj

$$\begin{tikzcd}
 & & M': \{..Cj..\} \arrow[dll] \arrow[dl] \arrow[dr] & \\
E3 & N': \{............\} \arrow[d] & & N'': \{...........\} \arrow[d] \\
 & E1 & & E2
\end{tikzcd}$$

Fig 3.3 Move-cj Transformation

- Unification

Identical instructions are moved up from multiple successor nodes to a single node (Fig 3.4a).

$$\begin{tikzcd}
 & M: \{....\} \arrow[dl] \arrow[dr] & \\
N: \{..X..\} & & P: \{..X..\}
\end{tikzcd}$$

is transformed to:

$$\begin{tikzcd}
 & M: \{..X..\} \arrow[dl] \arrow[dr] & \\
N: \{....\} & & P: \{....\}
\end{tikzcd}$$

Fig 3.4a Unification Transformation

Copies are required if any of the nodes which contained the moved instruction have multiple entries (Fig 3.4b).

$$\begin{tikzcd}
{} \arrow[dr] & & M: \{....\} \arrow[dl] \arrow[dr] & & {} \arrow[dl] \\
& N: \{..X..\} & & P: \{..X..\} &
\end{tikzcd}$$

is transformed to:

$$\begin{tikzcd}
& M: \{..X..\} \arrow[dl] \arrow[dr] & & {} \arrow[d] & {} \arrow[d] \\
N: \{....\} & & P: \{....\} & N': \{..X..\} & P': \{..X..\}
\end{tikzcd}$$

Fig 3.4b Unification Transformation

These core transformations can form the basis of a wide variety of global scheduling algorithms. In addition, they can be used as the basis of more complex transformation primitives. However, register renaming is not considered in Percolation Scheduling.

Ebcioglu [Moon and Ebcioglu 92] has developed a scheduling technique called Enhanced Percolation Scheduling in conjunction with the IBM VLIW machine at the IBM Watson Research Center in New York to improve the scheduling of loops. This method develops percolation scheduling to allow code motion across the loop back edge. It also adds the renaming mechanism. The loop is composed of a number of instructions termed "tree instructions" by Ebcioglu. Initially each tree instruction is composed of one instruction plus a jump to the following tree instruction. Thus if a loop contains five tree instructions labelled from L1 to L5 and the instruction L1 can be percolated up into L5 some of the operations from iteration M + 1 will be executed in parallel with the operations from iteration M. If the percolation scheduled loop contains N instructions, the multiple successive loop iterations can be overlapped by percolating instructions across the loop back edge N times. Enhanced Percolation Scheduling automatically generates a software pipelined loop of arbitrary complexity. Loop prologues and epilogues are generated automatically during code percolation.

Ebcioglu compared VLIW SPEC benchmark performance with the IBM RISC System/6000. Four integer SPEC benchmarks were used. The C code was compiled using a PL.8 C-compiler and then parallelised to VLIW code. The VLIW processors was assumed to have a data and instruction cache of 256kbytes and 4096 VLIW words respectively. Eight cycles penalty for data cache miss and 15 cycles penalty for an instruction cache miss were also assumed. Each operation is assumed to take a single cycle. A speedup of 3.7 is obtained when compared to the IBM RISC System/6000. Although this is an impressive speedup, Ebcioglu's model is unrealistic in two important respects. First, there are no branch delays in the model. Instead each VLIW instruction contains all possible next addresses. Second, the results assume 16 operations per LIW with up to 16 ALU operations, 16 branch operations and 8 memory operations. Therefore, as with the work by Wall and others outlined in Section 3.1 unlimited resources leads to very impressive figures.

In addition to the scheduling techniques described above, there are also scheduling techniques based on branch prediction. Branches constitute between 15 and 30% of instructions executed on a typical machine and severely restrict an instruction scheduler's ability to maximise instruction parallelism.

In general, there are two ways to predict branches. These are hardware branch prediction and software branch prediction. They both achieve similar prediction accuracy. However, hardware branch prediction schemes predict branches dynamically at run time and rely on the outcome of a branch being stable over a period of time. Software branch prediction uses either a static or dynamic approach to predict branch directions at compile time. In a static approach, prediction is based on the likely outcome of each branch instruction format. For example, Smith [Smith 81] found that "branch if negative", "branch if equal" and "branch if greater than or equal" were usually taken. Therefore, these branches were always predicted to be taken. Other conditional branches were always predicted not to be taken. Also, a backward branch to a loop header was always predicted to be taken. Although a simple technique, Smith claimed a prediction accuracy from 65.7% to 99.4% with a mean of 86.7%. The problem with this approach is that it relies heavily on the characteristics of branches within specific workloads, and these characteristics tend to vary from workload to workload.

In the dynamic approach, branch information is gathered by profiling a program [Hwu 89]. First, the program is compiled into an executable intermediate form with probes inserted at the entry to each basic block. The program is then run one or more times using a representative input suite and the probes are used to gather statistics on the outcome of each branch instruction. These statistics can then be used to guide the instruction scheduler.

Alternatively, the scalar instruction set can be modified to include a prediction bit in the branch instruction format. This bit is set by the compiler using the branch prediction bits to prefetch instructions from the most likely paths. However, since branch prediction is imperfect it generates additional instruction fetch traffic and increases the instruction cache bandwidth requirements.

Branch prediction is useful for all scheduling methods but is crucial for trace scheduling and boosting. It helps the other methods to make the best use of scarce resources by directing the scheduler to schedule the most likely paths first.

4. THE HARP PROJECT

iHARP [Steven 92a] is a VLSI VLIW processor. It has recently been fabricated and is currently undergoing tests. A range of HARP architectures is defined in Section 4.2 for use in this investigation and is based on the iHARP VLSI processor.

4.1 iHARP Processor

iHARP provides 32, 32-bit general purpose registers and eight 1-bit boolean registers. Four parallel pipelines and a memory unit share the general purpose register file. The memory unit is multi-ported with two write and ten read ports. It is implemented physically as five two-port register files, with common write ports and separate read ports. The two write ports are controlled by register writeback logic for instructions executed conditionally.

iHARP uses a four-stage pipeline. During the IF stage iHARP fetches a 128-bit LIW from the instruction cache. In the RF stage, the four instructions are decoded in parallel by four decoders and register operands are fetched from the register file. In the ALU/MEM stage the four instructions are dispatched to four functional units for concurrent execution. During the WB stage all computational results are written to the general purpose register file.

The iHARP instruction set is shown in Fig 4.1 and the short instruction format in Fig 4.2. Integer division and multiplication are performed by multiple integer step instructions. iHARP also provides an instruction that is a combination of a separate shift and ALU instruction. Both conditional execution and combined instructions are among those features evaluated.

The functionality of iHARP's four pipelines is shown in Fig 4.3. Each field in a LIW can specify several different short instructions. Therefore, the instruction decoders must issue and dispatch instructions to the appropriate functional unit. iHARP is a real processor and therefore resources are limited. Therefore the implementation of the four pipelines requires careful consideration in deciding which pipeline is to execute which functions. Both pipelines one and three can execute branch instructions. However, when two branches are executed in parallel, the branch instruction in pipeline three has priority. A short instruction with a long literal must occupy two successive words in the instruction memory. Therefore, such instructions must start in either pipeline zero or pipeline two allowing the literal to be found in pipeline one or pipeline three. Shifts are not supported uniformly in all four pipelines. Pipelines zero, one and two all support left-shift operations of one, two and four bits plus right-shift operations of one, two, three and four bits. Pipeline three supports left-shift operations of one, three, four and eight bits and right-shift operations of five, six and eight bits.

4.2 The HARP Family

Each member of the HARP family issues a different number of instructions in parallel. The number issued varies from one to five short instructions per LIW. The architecture with a one-instruction issue rate is referred to as the HARP RISC. This model is used as a base architecture and all performance measurements are relative to the performance of this model.

The HARP RISC adopts the instruction set shown in Fig 4.1 and employs the register files, addressing modes and four pipeline stages used in iHARP. Functional units include a decoder, an ALU, a shifter, an integer step unit, a PC unit, a boolean unit and an address unit. Complete register bypassing is provided and three parallel read ports and one write port are provided on the general purpose register file.

Table 4.1 shows the variations based on the HARP RISC model for a two-instruction issue, three-instruction issue, four-instruction issue and five-instruction issue HARP model. All but the HARP RISC provide two address units enabling two memory reference instructions to be executed providing they are executed on mutually exclusive boolean values. Two variations of these models are also considered. The first allows two memory reference instructions to be executed in parallel by providing two independent 32-bit wide paths between the CPU and a two-port data cache. The second investigates the impact of allowing two branch instructions to be executed in parallel. Support is also provided for speculative execution.

4.3 Compiler and Scheduler

This section presents a brief overview of the software used during the course of this investigation.

4.3.1 iHARP Compiler

Stanford benchmarks were compiled using a C compiler constructed using leading public domain software, GNU CC version 1.30. Code was generated for the iHARP processor. The code produced was then scheduled by RLS a resource limited scheduler. It is a resource limited scheduler in the sense that code motions resulting in the duplication of resources are limited. The scheduled code was then executed on the HARP simulator [Whale 92].

4.3.2 RLS a resource limited scheduler

RLS, a resource limited scheduler, is used to exploit the fine-grained parallelism inherent in non-numeric programs. RLS allows the instructions belonging to more favoured basic blocks to consume hardware resources first. The topology of the flow graph and a set of scheduling heuristics determine these scheduling precedences. In contrast many approaches take no account of limited hardware resources. VLIW architectures inevitably expand object code size since in practice it is impossible to fill every long instruction with useful short instructions. RLS aims to control the number of long instructions generated thereby giving a higher cache hit rate than many other scheduling algorithms which produce many more long instructions are able to do. RLS does not require branch profiling so that it is more flexible when applied to commercial or non predictable user applications.

Conceptually, RLS consists of high level and low level code transformations which are performed at the procedural level. The high level code transformation maintains overall control of the scheduling process while the low level code transformation is performed at the basic block level.

The high level directs the low level code transformations and rearranges the flow graph each time a low level code transformation is complete. The high level transforms the scalar instruction of a procedure into a linked data structure, constructs a flow graph, detects loops, unrolls loops, and selects the next basic block for the low level code transformations. Instructions in the basic blocks are then moved in topological order into the instruction window. Instructions are moved towards the top of the flow graph while respecting data dependencies and resource constraints.

Low level code transformation is complete when all instructions in the basic blocks have been scheduled. High level code transformation is complete when all basic blocks in a procedure have been selected for scheduling. RLS is complete when all procedures in the program have been scheduled. The resultant window is a single stream of parallel code which can be executed on a specific multiple instruction issue processor.

5. THE ARCHITECTURAL FEATURES OF iHARP EVALUATED

This section presents and discusses the results of the evaluation of the architectural features under investigation. The architectural features under investigation are those features outlined in the introduction. The purpose of obtaining these results is to show how these features can affect the performance of a multiple instruction issue processor. The cache hit rate is assumed to be 100% in all cases.

The single pipeline scheduler attempts to fill branch delay slots with useful instructions. Fig 5.1 shows the number of cycles required to execute eight Stanford benchmarks on the HARP RISC before and after the delay slots have been filled. Performance improved by 7.3% on average (harmonic mean), after the branch delay slots had been filled. All the results presented for the investigation of the architectural features use the HARP RISC after, and not before, the branch delay slots have been filled as the base measurement which could otherwise show deceptive parallelism.

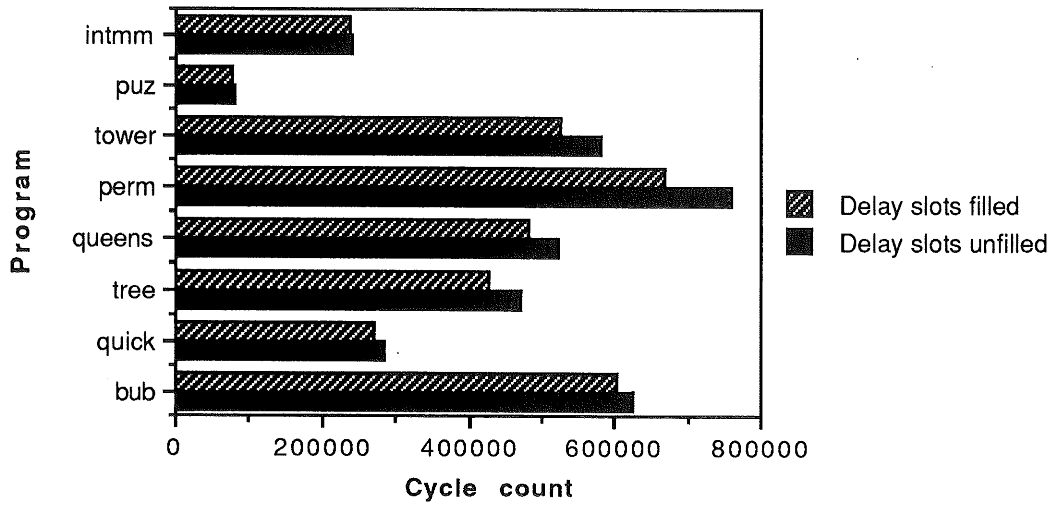


Fig 5.1 HARP Scalar RISC dynamic cycle counts
Before and after filling delay slots

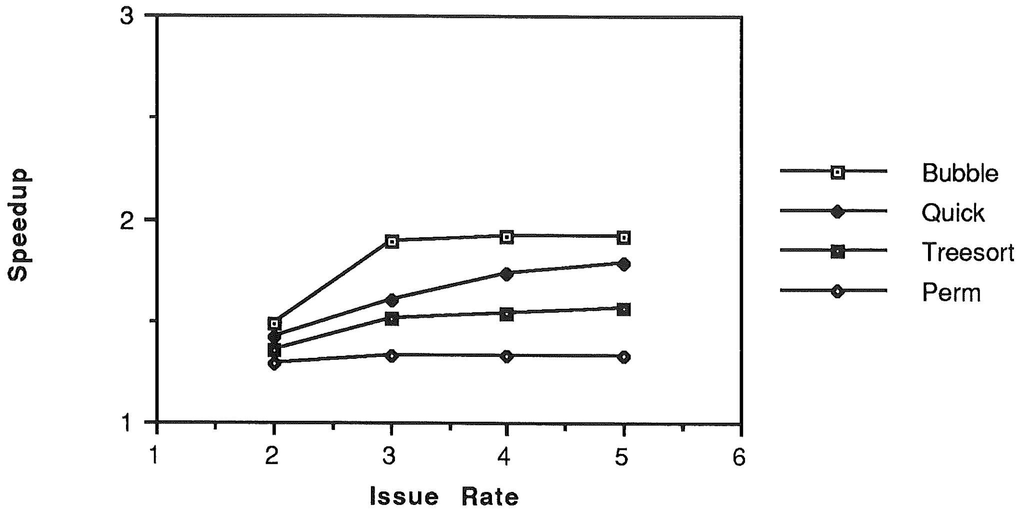
5.1 Instruction Issue Rate

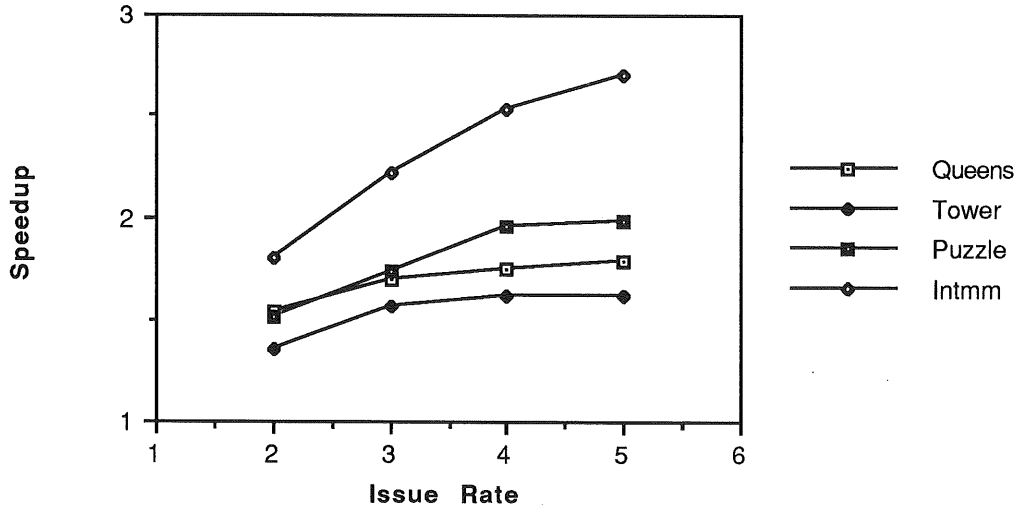
A MII architecture's performance is to some extent dependent upon the number of instructions it can issue for execution in each cycle. However, there are limitations and costs associated with the issue rate. These are limited parallelism of the source code, limited hardware resources and code size increase. Limited parallelism can be overcome by a smart scheduler unrolling loops and other such similar techniques. Limited hardware resources can be overcome by duplicating functional units. However, this will require an increase in the memory and register file bandwidth. Also, keeping these functional units busy requires enough work in the LIW so that code size does not increase too excessively. This leads to the third point of code size increase since NOPs in unused instruction slots and loop unrolling will increase code size. Thus considering all of the above points makes it interesting to find out the most cost effective issue rate.

The adeptness of RLS's parallelising capabilities was measured by comparing the cycle count and code size generated for both serial and parallel code. The serial code was that produced by the single pipeline scheduler. The parallel code was produced by the different configurations of the HARP RISC discussed in Section 4, namely, two instruction issue, three instruction issue, four instruction issue and five instruction issue processors. In all cases one memory port was assumed. Tables 5.2-5.5 show the speedup obtained over the original serial code. Fig 5.2 shows the average speedup obtained over the original serial code. The speedup is 1.45 for an issue rate of two and 1.66, 1.74 and 1.76 for issue rates of three, four and five respectively.

There is a significant speedup as the issue rate increases from two to four and a slight speedup as the rate increases from four to five. Speedup then begins to level off. The speedup obtained over the HARP RISC with the delay slots filled is 1.76 for five pipelines and 1.74 for four pipelines. These figures compare favourably with other groups working in the area. The IMPACT group [Chang 91] obtained a speedup of 1.6 for an issue rate of two and 2.00 for an issue rate of four. The group at Stanford working on boosting [Smith et al 92] achieved a speedup of 1.24 for their base superscalar with an issue rate of two and no boosting, 1.45 for their base superscalar using a boosting level of one and 1.5 for their base superscalar using a boosting level of three. Although the figures for HARP are slightly lower than for the IMPACT group the speedup for HARP would in fact be 1.6 for two and 2.00 for four if the base measurement was for HARP before the delay slots were filled. IMPACT do not say what the base model is i.e. delay slots filled or unfilled. iHARP is a real processor which has been fabricated and therefore has a limited number of resources. Therefore, 1.8 is a good result and does compare well with other research groups which tend to use a paper design and unlimited resources. For an issue rate of four a speedup of two is probably the best that can be expected, that is, 50% of the pipelines working each cycle.

Figs 5.3 and 5.4 shows the code size increase as the issue rate increases. At an issue rate of two code size increases by 1.38. At higher issue rates the increase is 1.86, 2.34 and 2.82 for issue rates of three, four and five respectively. This emphasises the main problem of VLIW architectures that of code explosion. NOPs are inserted in any unused instruction slots in the long instruction word and as the amount of parallelism obtained by the scheduler reaches a limit so the percentage of NOPs increases dramatically. Fig 5.5 shows the percentage of NOPs in each program. For example, the percentage of NOPs in the program perm increases from 36% for an issue rate of two to 70% for an issue rate of five. Fig 5.6 shows that the average increase in the percentage of NOPs is from 33% for an issue rate of two to 65% for an issue rate of five, a two-fold increase. The problem of code explosion is addressed in superscalar architectures which do not insert NOPs in instruction slots. Therefore code size does not increase with increased issue rate. The computer architecture group at the University of Hertfordshire are developing a minimal superscalar processor to address this problem.





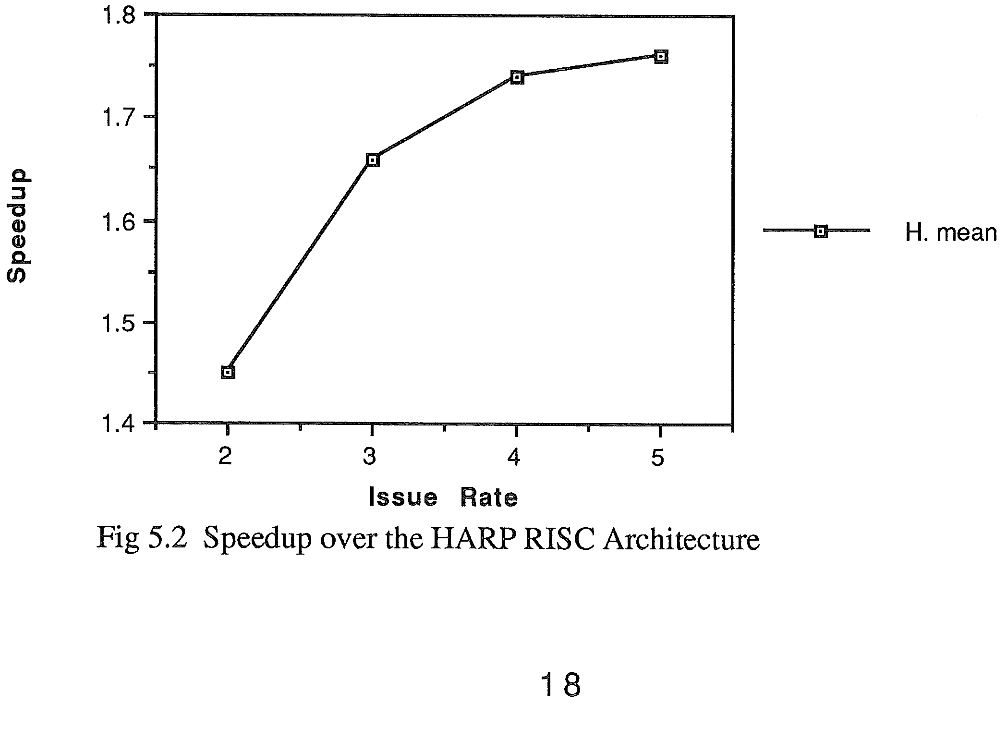
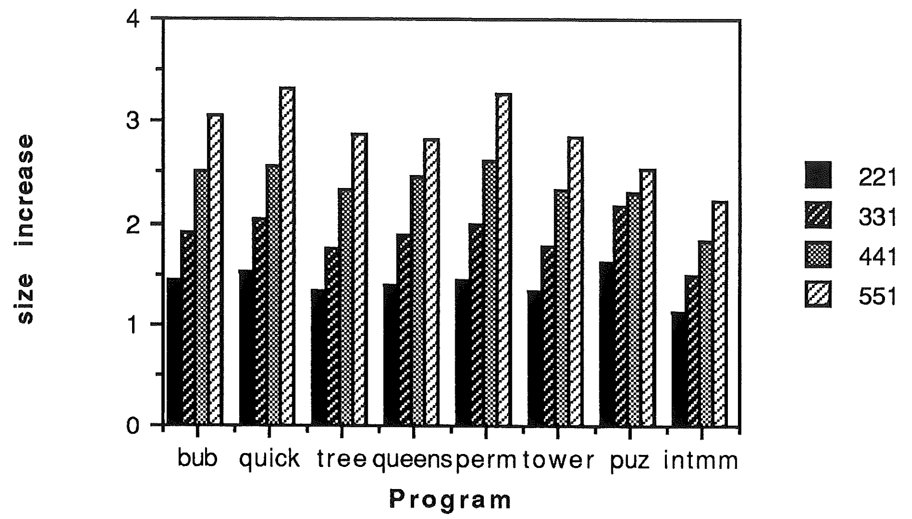


Fig 5.2 Speedup over the HARP RISC Architecture



221 - Issue rate of two, two writebacks and one memory port.
331 - Issue rate of three, three writebacks and one memory port.
441 - Issue rate of four, four writebacks and one memory port.
551 - Issue rate of five, five writebacks and one memory port.

Fig 5.3 Ratio of Code Size Increase over the HARP RISC Architecture

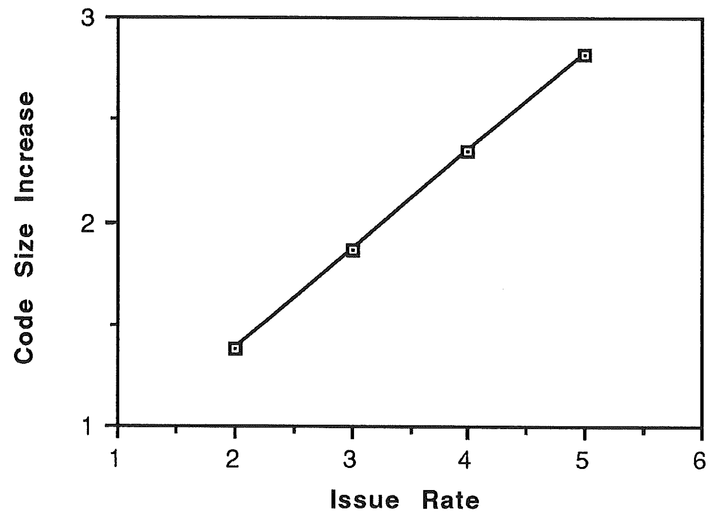


Fig 5.4 Average ratio of Code Size Increase over the HARP RISC

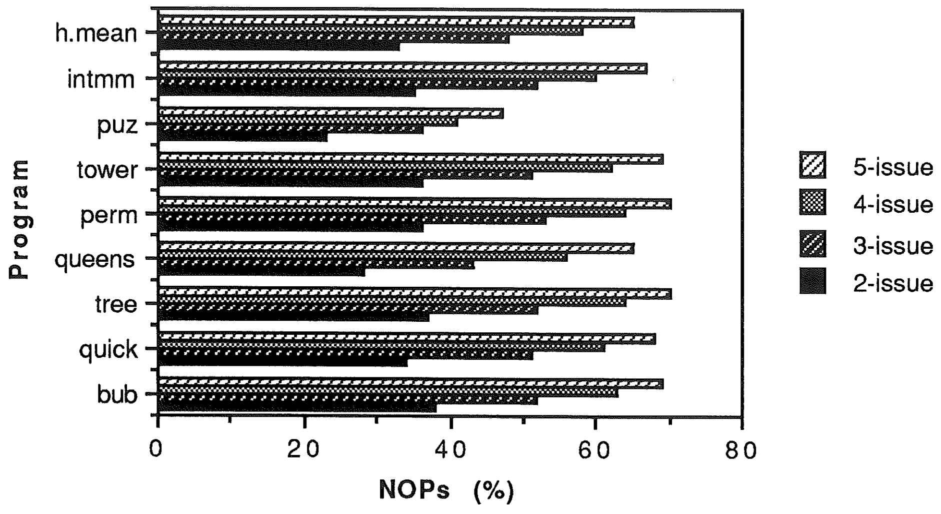


Fig 5.5 Percentage of NOPs in each benchmark

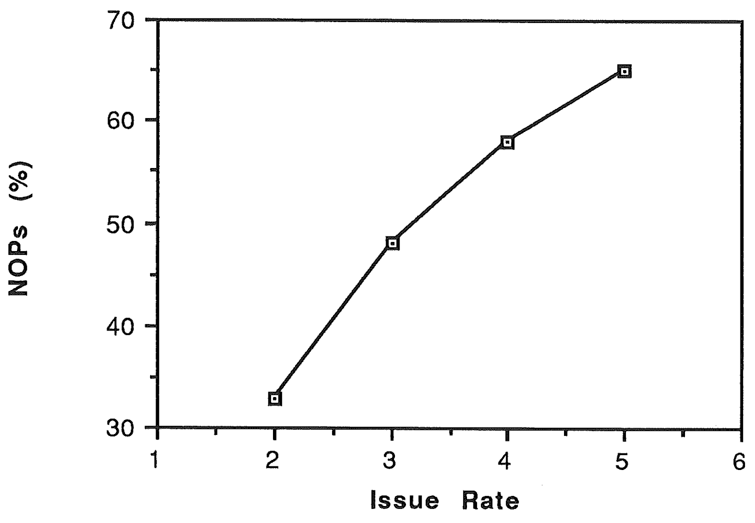


Fig 5.6 Average Percentage of NOPs

5.2 The number of cache memory ports

Traditionally only one memory port is available to microprocessors to access data. A second data cache memory port is added which gives enough capacity to do two parallel reads or writes each cycle. Thus, the impact on performance of using two data cache memory ports was investigated for issue rates from two to five. Tables 5.6-5.9 shows the speedups obtained for individual benchmarks. Fig 5.7 shows the average speedup obtained for all eight benchmarks for both one memory port (1M) and two memory ports (2M). The speedup obtained ranges from 1.51 for an issue rate of two to 1.96 for an issue rate of five.

Low parallelism realised in programs can, in part, be attributed to a high percentage of memory reference instructions. Several programs could be said to be memory intensive. These are identified as perm, bubble, queens and tower. These programs have benefited significantly from the extra data cache port. Fig 5.7 shows that a performance increase is obtained at all issue rates. For example at an issue rate of four performance improves by approximately 10% when two cache ports are available.

The cause of the speedup will be in part due to the fact that two loads can now take place in parallel. Therefore, more data is available for use each cycle and therefore more useful work can also be undertaken each cycle. Thus it would be expected that a performance improvement would be obtained which is indeed the case.

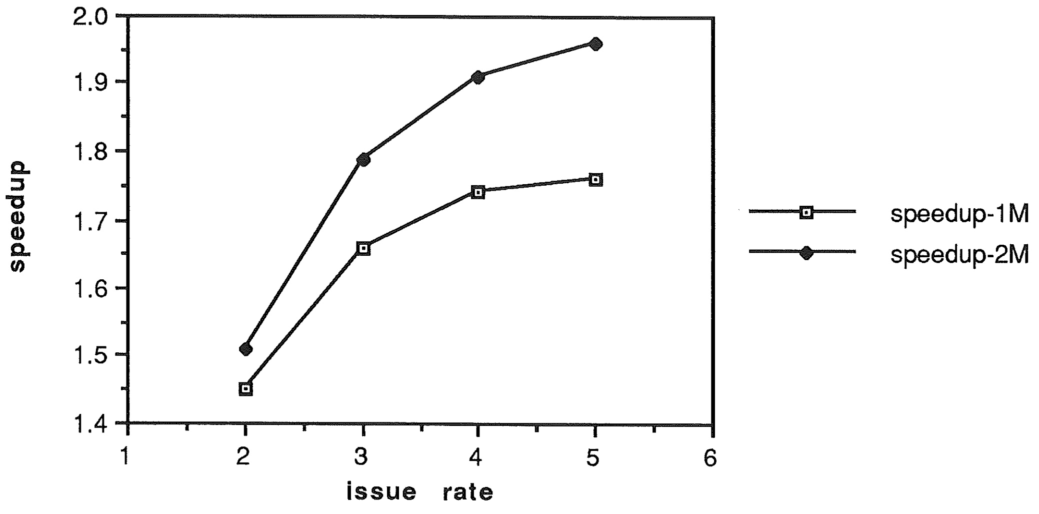


Fig 5.7 Speedup versus the number of data cache ports

5.3 Conditional execution

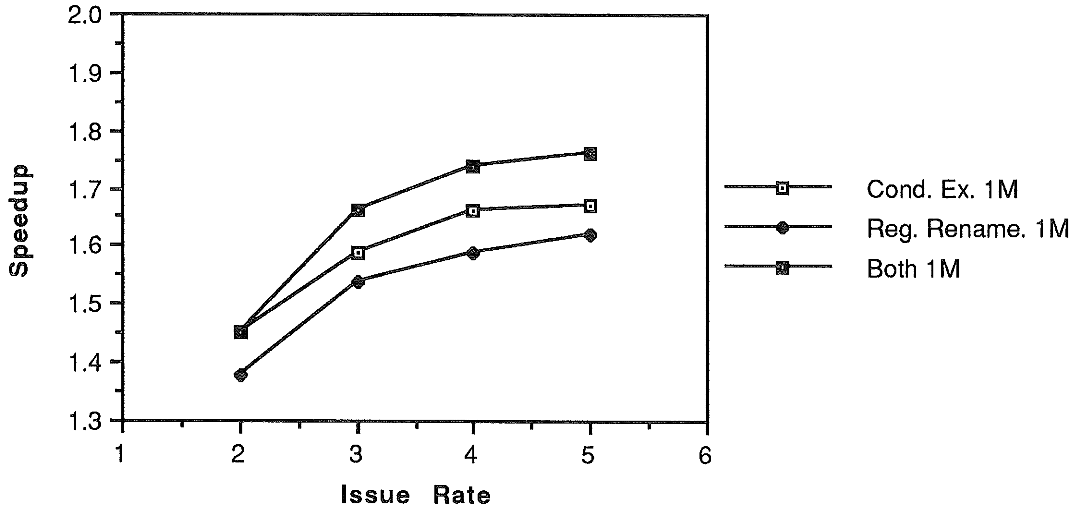
Conditional execution has been proposed by a number of people including Hsu and Davidson [Hsu 86], Acorn ARM processor [Furber 89] and Moon and Ebcioglu [Moon 92]. However, little if any evaluation of its benefits has been carried out. Therefore, scheduling is compared with both register renaming and conditional execution, conditional execution only and register renaming only to establish the effectiveness of conditional execution on performance.

Tables 5.19-5.42 and Fig 5.8 shows the results obtained using the various combinations of register renaming and conditional execution for both one and two memory ports. For one memory port the speedup obtained for both register renaming and conditional execution is 1.45 for an issue rate of two increasing to 1.76 for an issue rate of five. Using conditional execution only the speedups are 1.45 to 1.67 for issue rates from two to five. With register renaming only the speedups are 1.38 to 1.62 for issue rates from two to five. When two memory ports are considered an improvement in the speedup is obtained. Thus, using both register renaming and conditional execution the speedups are 1.51 to 1.96. Using register renaming only the figures are then 1.51 to 1.82. Finally, for register renaming only the figures are 1.44 to 1.70. Thus, at an issue rate of two conditional execution and a combination of both conditional execution and register renaming improves performance by 5% when compared with register renaming only for both one and two memory ports. At higher issue rates a combination of both methods improves performance significantly when compared with register renaming whilst conditional execution lies somewhere in between. In fact the combination is still increasing at an issue rate of five suggesting that an examination of higher issue rates is necessary. The surprising result is that conditional execution improves performance more than register renaming. It would be expected that register renaming would show a greater speedup than conditional execution. This result is probably a consequence of the scheduling algorithm used which is designed to make very good use of the conditional execution feature.

The improvement in performance when both methods are used shows the benefit of using conditional execution as well as register renaming.

There are advantages and disadvantages associated with both methods. The advantages of register renaming are twofold. First, it is more flexible and supports code motion across multiple branches. Second, it can be used to resolve anti and output data dependencies. However, its disadvantage is that it introduces restoring code which consumes extra hardware resources. This, in turn may prevent further code motion. The advantages of conditional execution are fourfold. First, it conserves resources in that restoring code is not required. Second, live variable analysis is not required. Third, it reduces the number of registers used. Fourth, it avoids the requirement for speculative execution. There are two main disadvantages. First, it requires a boolean guard which introduces a dependence between the boolean guard and the instruction being moved thereby limiting code motion. Second, it cannot resolve anti and output data dependencies.

Thus, both methods have advantages and disadvantages. However, a better performance is produced when both methods are used together than when either is used separately. Therefore, both methods should be incorporated into future instruction schedulers.



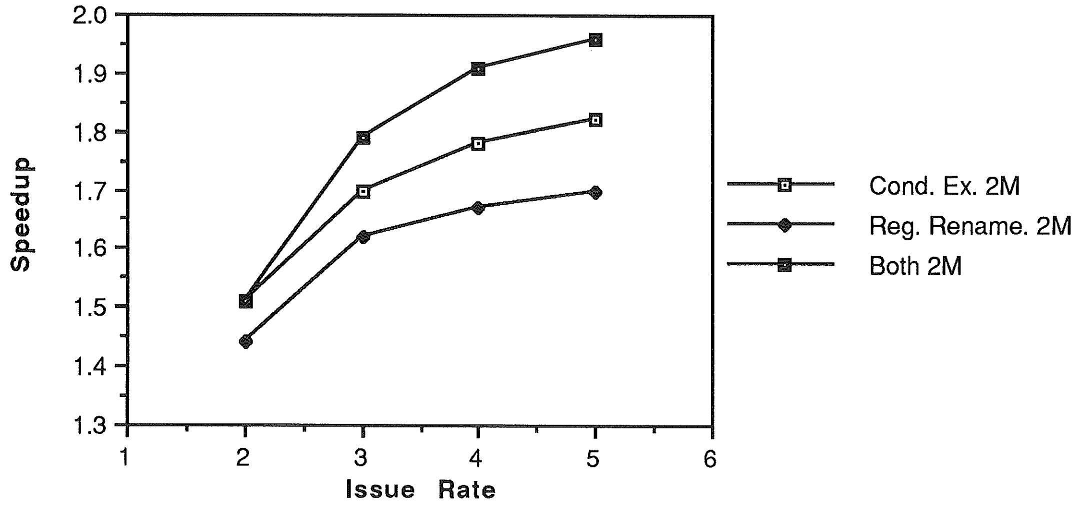


Fig 5.8 Impact of conditional execution and register renaming for both one and two memory ports

5.4 ORed Indexing

A RISC processor typically requires five pipeline stages to execute a load instruction [Hennessy 90]:

IF:	Instruction Fetch.
RF:	Register Fetch.
ALU:	Memory address calculation.
MEM:	Data cache access.
WB:	Write result to register file.

Since the data accessed from the cache is not available until the end of the MEM stage, it can only be used by the next instruction if the pipeline is stalled for one cycle. The load delay is therefore one.

In contrast, iHARP uses a four stage pipeline as outlined in Section 4.1:

IF:	Instruction Fetch.
RF:	Register Fetch.
ALU/MEM:	Perform operation or access data.
WB:	Write result to register file.

Two changes have been made to the original RISC pipeline. First, the address calculation has been moved to the RF stage. Second, the ALU and MEM stages have been combined. This second change is possible because loads and stores no longer use the ALU to compute memory addresses. The major advantage is that the result of a load operation is now available at the end of the ALU/MEM stage and can therefore be bypassed directly to the next sequential instruction with no load delay.

The key to moving the address compuation to the RF stage is to reduce the computation to a bitwise logical OR between the two address components [Steven 88]. A logical OR is equivalent to an addition if no carries are generated. This condition is met if the address components never have a logical one in the same bit position.

In order to use a logical OR in address computations, the compiler ensures that the bottom n bits of the stack pointer are always zero by aligning the stack pointer on a power of two memory address boundary. A stack offset can then be added safely to SP using a logical OR. Glew [Glew 89] suggested the term ORed indexing to describe this addressing mechanism.

iHARP's addressing mechansims of offset(Ri) and (Ri,Rj), where Ri and Rj can be any general-purpose register and since R0 is always zero register indirect and direct addressing are also available, are compared with three other addressing mechanisms for both serial and parallel execution. The other addressing mechanisms are traditional RISC addressing modes with a five stage pipeline where the load delay is one, register indirect and direct addressing only where the four stage pipeline can be used thus avoiding a load delay, and register indirect, direct and stack pointer relative addressing where a dedicated address adder is required for adding the SP to an offset and where SP must be implemented as a register outside the general-purpose register file.

Performance in terms of cycle counts and dynamic code size is obtained for each addressing mechanism.

Execution times and dynamic instruction counts are shown in Tables 5.14-5.16. The relative performance of the addressing modes are shown in Fig 5.9.

The best execution times for serial code were obtained with ORed indexing and with a dedicated SP adder. Using traditional RISC addressing modes degraded performance by 3%, while using register indirect addressing degraded performance by 10%.

The move from serial to parallel code significantly changed the relative performance. With parallel code, ORed indexing, register indirect and a dedicated stack pointer adder all performed equally well, while using traditional addressing mechanisms degraded performance by 10%. The VIPER group [Abnous et al 92] found that register indirect addressing yielded a similar 8.4% performance improvement over traditional addressing modes in a VLIW environment.

In all cases the performance advantage over traditional addressing mechanisms is achieved by executing more instructions. Using ORed indexing or a dedicated SP address adder, 4% more instructions are executed, while with register indirect addressing 14% more instructions are executed.

Traditional addressing modes perform relatively well in a single pipeline because the compiler can usually hide the load delay by scheduling useful instructions in the load delay slot. In contrast, in parallel code any instruction which could be used to fill load delay slots can also be executed in parallel with the load instruction. As a result increasing the latency of load instructions has a greater impact on the execution time of parallel code.

In contrast register indirect addressing performs significantly better in a parallel environment. This improved performance is a direct result of a VLIW processor's ability to precompute addresses in parallel with other instructions. While these address computations increase the instruction count, the impact on performance is minimal.

While both ORed indexing and register indirect addressing are easily incorporated into a VLIW or superscalar design, the traditional addressing mechanisms are less easily accommodated. One approach is to replicate the five-stage pipeline described above. Unfortunately, delaying the ALU write back for one cycle significantly increases the number of bypassing paths required. A more attractive alternative is to allow ALU instructions to return their results to the register file in the fourth pipeline stage. However, since ALU and load instructions now write to the register file in different pipeline stages, an extra write port is required to avoid pipeline stalls. Alternatively, if an extra port is not provided, the variable instruction latency makes it more difficult for the instruction scheduler to minimise stalls by allocating write ports to instructions.

Although the use of a dedicated stack pointer adder produces excellent results, this mechanism is very dependent on the most common addressing mode being SP plus offset. In contrast, the other addressing modes treat all registers identically. The mechanism also scales very badly. For example, if parallel loads are required, a dedicated adder is required for each load. In general this mechanism is the least flexible of the alternatives examined and is therefore less attractive than the performance figures suggest.

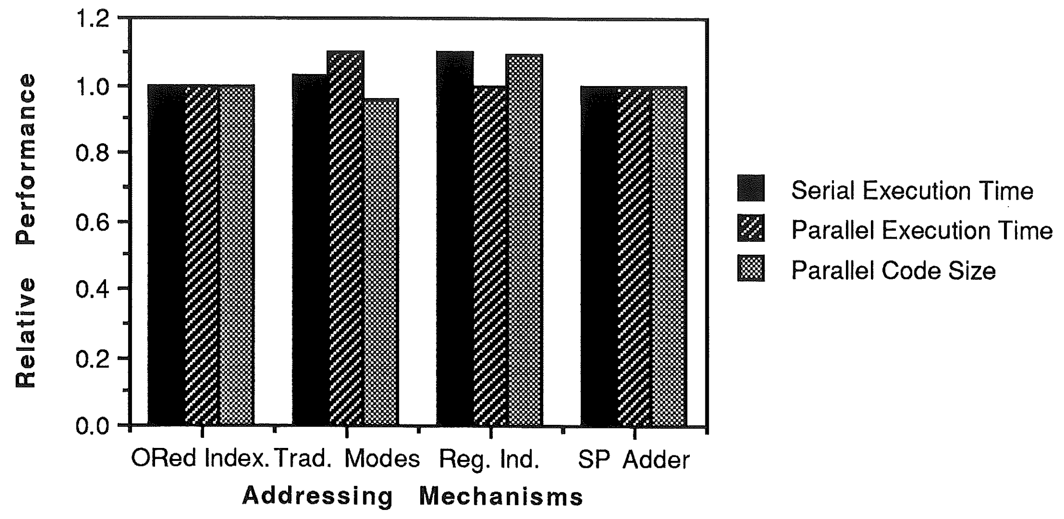


Fig 5.9 Relative performance of four different addressing mechanisms

5.5 The number of register writeback ports

Parallel writes to the general-purpose register file are essential in maintaining high execution rates in MII architectures. iHARP provides two writeback ports. Therefore, during each cycle a maximum of only two results can be returned to the register file. Thus, the writeback capability is varied between two, three and four writebacks to ascertain the effect this has on performance. As it is more complex implementing write ports than read ports the results will show whether the increase in hardware costs and complexity incurred in providing extra writeback ports can be justified.

Tables 5.10-5.12 and Fig 5.10 shows the impact on performance of varying the number of register writeback ports. In all cases one memory port and an issue rate of four is assumed. The speedups obtained are 1.66, 1.73 and 1.74 for two, three and four register writebacks respectively.

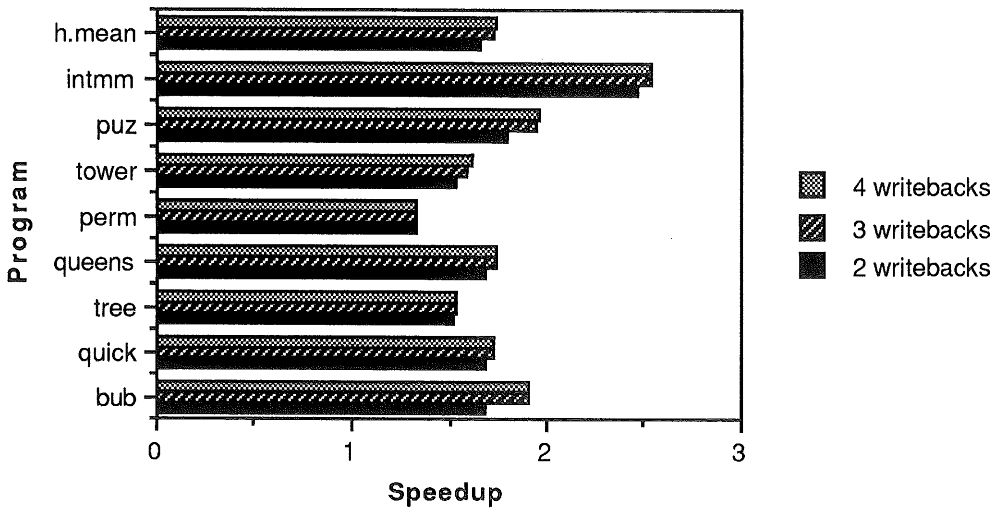


Fig 5.10 Speedup using a different number of register writebacks

Parallel writes to the general-purpose register file are essential to MII HARP architectures. iHARP's register file provides only two writeback ports. Therefore, during each cycle only two results can be returned to the register file. The figures show that increasing the number of register ports from two to three increases performance by approximately 4% while a further increase in the number of register writeback ports from three to four has a negligible effect. This suggests that in most cases not more than two or three instructions will be returning values to the register file in a four pipeline machine. Therefore, the fourth writeback port will not be used and the additional feature of a writeback bit could therefore only improve performance by 4%.

5.6 The number of branch units

iHARP is assumed to have two parallel branch units but since the extra cost of an additional branch unit is small many VLIW architectures provide multiple branch units [Ebcioglu 88][Fisher 83]. The impact of providing a single branch unit is investigated for a four instruction issue rate model to find out whether multiple branch units are actually necessary.

Table 5.13 and Fig 5.11 shows the speedup using one and two branch units. The results show that the performance impact was in fact negligible suggesting that most of the time only one branch is used per cycle.

Although the RLS scheduler allows two branches to be sheduled in parallel, code motion is always upwards. A branch at the end of a basic block will therefore only be scheduled in parallel with another branch if all the other instructions in the basic block have already been moved in parallel with a preceding basic block. Even when parallel scheduling of branches does occur there is no guarantee that execuing two branches in parallel will reduce the average execution time. The frequent occurrence of recursive subroutine calls in some benchmarks further reduces the opportunities to schedule branch instructions in parallel.

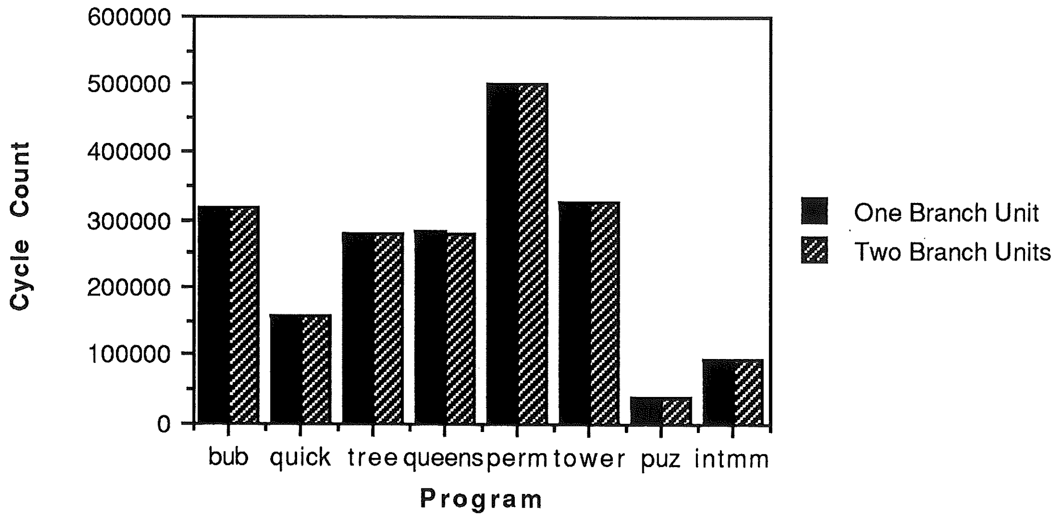


Fig 5.11 Speedup using one and two branch units

5.7 Combined instructions

iHARP provides instructions which are a combination of a shift instruction followed by an ALU instruction in source code. Combining was also proposed by Nakatani and Ebcioglu [Nakatani 89]. For example, the two separate instructions

ASL R6, R7, #6
ADD R5, R6, R2

could be combined to become

ADD R5, R7(ASL #6), R2

providing R6 is not live after the ADD instruction. Preliminary studies [Steven 92b] have found that 16% of instructions involving a shift can be combined in this way.

Tables 5.17-5.18 and Fig 5.12 show the results of the simulations for this feature. Table 5.17 shows the impact of instruction combining on sequential code. The speedup is almost identical for both sequential and parallel code at 1.02 and 1.03 respectively.

Different factors have played a part in the effect combining may have. In sequential code each instruction requires a cycle so combining shifts with another instruction essentially reduces the cycle count by one each time the sequence is executed. In parallel code however, without combining, a shift instruction and an ALU instruction could be scheduled in parallel thereby both being executed in the same cycle. However, if the instructions are then combined they will still be executed in the same cycle thereby incurring no savings or very little at all.

However, the conclusions are that too few instructions are combined to have a significant impact and is therefore not a critical factor in the speedup.

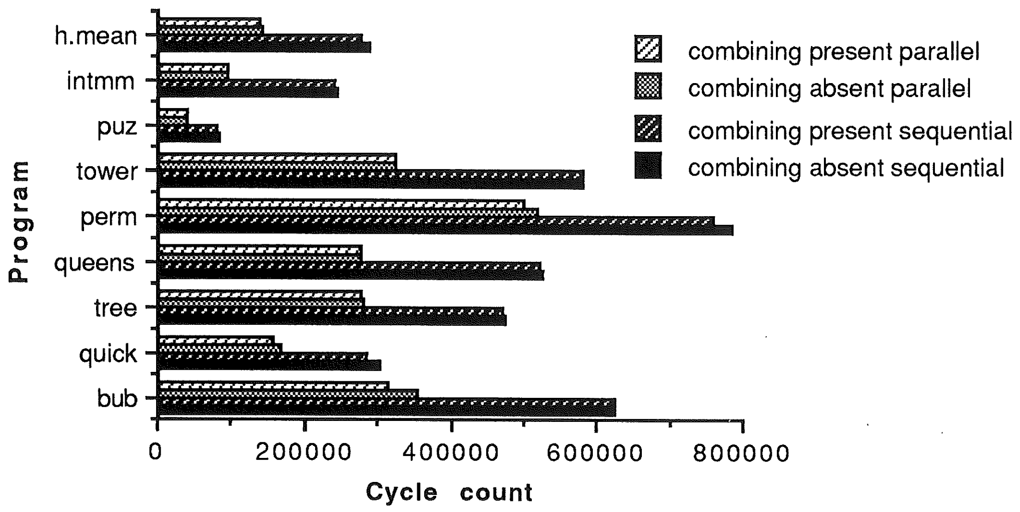


Fig 5.12 Impact of instruction combining on serial and parallel code

6. CONCLUSIONS

This paper has described RLS a resource limited scheduler which has been used to evaluate several architectural features of iHARP. This evaluation has been carried out using a HARP family of processors issuing two, three four and five instructions per cycle.

Speedup increases significantly as the issue rate increases from two to four and less significantly from four to five. This suggests that there will not be much impact on speedup if the issue rate is increased any further and therefore further development of the scheduling algorithm is required.

Both the number of register writeback ports and number of data cache ports has an affect on processor performance although the effect of the number of data cache ports is more significant. In contrast two parallel branch units do not significantly improve performance. This suggest that one branch unit is sufficient for a HARP processor. However, the recursive nature of the benchmarks may have removed some of the opportunities to schedule branch instructions in parallel.

Conditional execution has performed particularly well for both one and two memory ports. Register renaming has not performed as well as might be expected. However, this result is presumably because RLS has been designed to make good use of the conditional execution feature which has therefore been favoured. Interestingly, when both conditional execution and register renaming are combined it improves performance by approximately 5% for one memory port and approximately 8% for two memory ports when compared with conditional execution only.

Considering ORed indexing the results support the use of ORed indexing on HARP. A performance increase of 10% is achieved over traditional RISC addressing mechanisms while code expands by only 4%. Register indirect addressing performs equally well in a VLIW environment. However, code expansion is 14%. Finally, a dedicated stack pointer address adder performs as well as ORed indexing but is less flexible and involves greater hardware complexity.

The HARP instruction set provides for limited shifting to be performed before each ALU operation, thus allowing two scalar instructions to be combined into one complex operation. Although some speedup has occurred for both sequential and parallel code the impact obtained is not very significant and suggests that instruction combining is not a critical factor in processor performance.

It can be concluded that when scheduling for four pipelines with limited resources the best instruction execution rate that can be achieved is two, that is half the pipelines working each cycle.

ACKNOWLEDGEMENTS

The authors would like to thank the rest of the HARP team, in particular, Roger Collins, Corrie Elston, Rod Adams and Dave Whale from Computer Science and Paul Findlay, Brian Johnson and Dave McHale from Electrical Engineering. They would also like to thank Professor M Loomes, Dr S L Stott, J A Davis and Dr P Kaye for their support throughout the HARP project. The HARP project is supported by SERC Research Grant GR/F88018.

REFERENCES

Abnous A and Bagherzadeh N "Architectural Design and Analysis of a VLIW Processor", U. C. Irvine, Technical Report No. 92-79, October 1992

Butler M, Yeh Tse-Yu, Patt Y, Alsup M, Scales H and Shebanow M "Single Instruction Stream Parallelism is Greater than Two", Proceedings of the 18th Annual International Symposium on Computer Architecture, May 1991, pp 276-286

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", Proceedings of the 18th Annual International Symposium on Computer Architecture, Toronto, Canada, May 1991, pp 266-275

Collins R "A Simulator for the HSP Superscalar Processor", Internal Report, University of Hertfordshire, 1993

Colwell R P, Nix R P, O'Donnell J, Papworth D B and Rodman P K "A VLIW architecture for a trace scheduling compiler", Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems, October 1987, pp 180-192

Ebcioglu K "Some Design Ideas for a VLIW Architecture for Sequential-Natured Software", Proceedings of the IFIP WG 10.3 Working Conference on Parallel Processing, Pisa, Italy, April 1988, pp 1-21

Fisher J A "Trace Scheduling: a technique for global microcode compaction", IEEE Transactions on Computers, C-30, (7), July 1981, pp 478-490

Fisher J A "Very Long Instruction Word Architectures and the ELI-512", Proceedings of the 10th Annual Symposium on Computer Architecture, June 1983, pp 140-150

Fisher J A, Ellis J R, Ruttenberg J C and Nicolau A "Parallel Processing: A Smart Compiler and a Dumb Machine", Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction, SIGPLAN Notices, Vol. 19, No. 6, June 1984, pp 37-47

Furber S "VLSI RISC Architecture and Organization", Marcel Dekker, New York, 1989

Glew A "ORed Indexing", personal communication, April 1989

Gray S M "Code Generation for a Long Instruction Word Architecture", PhD Thesis, University of Hertfordshire, December 1991

Hennessy J L and Patterson D A "Computer Architecture: A Quantitative Approach", Morgan Kaufmann, San Mateo, California, 1990

Hsu P Y T and Davidson E S "Highly concurrent scalar processing", Proceedings of the 13th Annual Symposium on Computer Architecture, 1986, pp 386-395

Hwu W W, Conte T M and Chang P P "Comparing Software and Hardware Schemes for Reducing the Cost of Branches", Proceedings of the 16th International Symposium on Computer Architecture, April 1989, pp 224-231

Johnson M "Superscalar Microprocessor Design, Prentice Hall, 1991

Lam M S "Software pipelining: an effective scheduling technique for VLIW machines", Proceedings of the SIGPLAN 88 Conference of Programming Language Design and Implementation, Georgia, USA, June 1988, pp 318-328

Lam M S and Wilson R P "Limits of Control Flow on Parallelism", Proceedings of the 19th Annual International Symposium on Computer Architecture, Gold Coast, Australia, May 1992, pp 46-57

Landskov D, Davidson S, Shriver B and Mallett P W "Local Microcode Compaction Techniques", Computing Surveys, Vol. 12, No. 3, September 1980, pp 261-294

Moon S and Ebcioglu K "An Efficient Resource-Constrained Global Scheduling Technique for Superscalar and VLIW Processors", Micro 25, Portland, Oregon, December 1992, pp 55-71

Nakatani T and Ebcioglu K "Combining as a Compilation Technique for VLIW Architectures", The 22nd Annual International Workshop on Microprogramming and Microarchitecture, Dublin, August 1989, pp 43-55

Nicolau A "Uniform Parallelism Exploitation in Ordinary Programs", Proceedings of the International Conference on Parallel Processing, August 1985, pp 614-618

Smith J "A Study of Branch Prediction Strategies", Proceedings of the 8th Annual International Symposium on Computer Architecture, May 1981, pp 135-148

Smith M D, Horowitz M and Lam M "Efficient Superscalar Performance Through Boosting", Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems, October 1992, pp 248-259

Steven G B "A novel effective address calculation mechanism for RISC microprocessors", SIGARCH, 16, (4), 1988, pp 150-156

Steven G B, Adams R G, Findlay P A and Trainis S A "iHARP: a multiple instruction issue processor", IEE Proceedings-E, Vol. 139, No. 5, September 1992, pp439-449[a]

Steven F L "A Preliminary Study of Combining Instructions", Internal Report, University of Hertfordshire, 1992[b]

Steven G B "The Hatfield Superscalar Architecture", Internal Report, University of Hertfordshire, 1993

Wall D W "Limits of Instruction-Level Parallelism", Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV), April 1991, pp 176-188

Wang L "Instruction Scheduling for a Family of Multiple-Instruction-issue Architectures", PhD Thesis, University of Hertfordshire, expected January 1994

Whale D J "Development of a Processor Simulation for iHARP", Division of Computer Science, University of Hertfordshire, April 1992

Fig 4.1 iHARP Instruction Set

Computational

ADD	Rdst,src1,src2
ADDV	Rdst,src1,src2
ADDC	Rdst,src1,src2
SUB	Rdst,src1,src2
SUBV	Rdst,src1,src2
SUBC	Rdst,src1,src2
AND	Rdst,src1,src2
OR	Rdst,src1,src2
EOR	Rdst,src1,src2
BIC	Rdst,src1,src2
DSTEP	Rdsta:Rdstb,Rsrc1a:Rsrc2a,Rsrc2b
MSTEP[i]	Rdst,src1,src2
MSTEPV[i]	Rdst,src1,src2

Memory Reference

LD Rdst,<ea>
LDB Rdst,<ea>
LD Bdst,<ea>
ST <ea>,Rsrc3
STB <ea>,Rsrc3
ST <ea>,Bsrc3

Special Purpose

MOV Rdst,SPR
MOV SPR,src2
EI
DI
TRAP #n,Bsrc3

Relational

GTS	Bdst,Rsrc1,src2
GES	Bdst,Rsrc1,src2
LTS	Bdst,Rsrc1,src2
LES	Bdst,Rsrc1,src2
GTU	Bdst,Rsrc1,src2
GEU	Bdst,Rsrc1,src2
LTU	Bdst,Rsrc1,src2
LEU	Bdst,Rsrc1,src2
EQ	Bdst,Rsrc1,src2
NE	Bdst,Rsrc1,src2

Boolean

AND	Bdst,Bsrc1,Bsrc2
OR	Bdst,Bsrc1,Bsrc2
EQ	Bdst,Bsrc1,Bsrc2
NE	Bdst,Bsrc1,Bsrc2
GT	Bdst,Bsrc1,Bsrc2
LT	Bdst,Bsrc1,Bsrc2
LE	Bdst,Bsrc1,Bsrc2
GE	Bdst,Bsrc1,Bsrc2

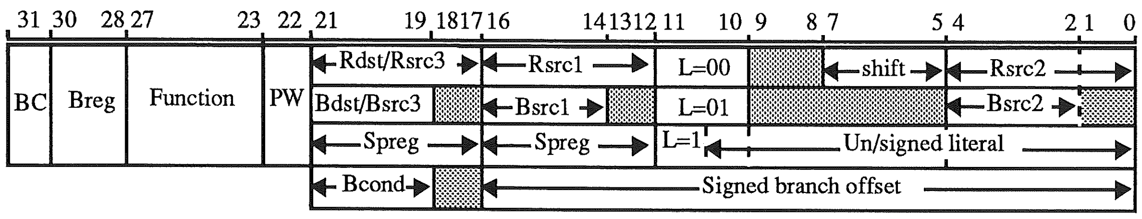
Branch

BT	Bsrc3,label
BF	Bsrc3,label
BSR	Rdst,label

Notes

Rsrc1, Rsrc2, Rsrc3:	32-bit, general-purpose source registers.
Rsrc1a, Rsrc2a, Rsrc1b:	32-bit, general-purpose source registers.
Rdst, Rdsta, Rdstb	32-bit, general-purpose destination registers.
src1 :	Rsrc1 optionally shifted.
src2 :	Rsrc2, 11-bit literal(signed) or 32-bit literal.
<ea>:	Rsrc1 ORed with src2
Bsrc1, Bsrc2, Bsrc3:	Boolean source registers
Bdst:	Boolean destination register
SPR:	PC or Status Register
i:	Specifies bit in Multiplication Register

Fig 4.2 iHARP Short Instruction Format



Key

BC	Boolean for conditional execution
Breg	Boolean reg. to test for conditional execution
Function	Instruction function code
PW	Permit write-back to register file
Rdst/ Rsrc3	General purpose destination register / store source
Bdst/ Bsrc3	Boolean destination register / store source
Rsrc1/ Bsrc1	General purpose/Boolean register containing 1st operand

Rsrc2/ Bsrc2		General purpose/Boolean register containing 2nd operand
L=	01	Inter-register format
	00	Long-literal format (32-bits)
	1X	Short-literal format
Shift		Shift control code
Spreg		Special purpose register. Used for Load and Store Special Purpose Register
Bcond		Second Boolean Condition for TRAP and Branch
▨		Bits not used in some formats

Fig 4.3 Functionality of iHARP pipelines

	Pipeline0	Pipeline1	Pipeline2	Pipeline3
1.	computational	computational	computational	computational
2.	relational	relational	relational	relational
3.	memory reference	-	memory reference	-
4.	-	-	boolean	-
5.	-	branch & return	-	branch & return
6.	-	special purpose	-	-
7.	-	-	-	traps
L.	-	32-bit literal	-	32-bit literal

Note

Instructions in pipeline 0 and 1 can use a 32-bit literal from an adjacent pipeline.

TABLES

Table 2.1 Functional Units provided on the HARP Family of processors

	HARP RISC	2-inst Issue	3-inst Issue	4-inst Issue	5-inst Issue
Issue Rate	1	2	3	4	5
Decoder	1	2	3	4	5
ALU	1	2	3	4	5
Shifter	1	2	3	4	5
Integer Step Unit	1	2	3	4	5
PC Unit	1	1	1	1	1
Address Unit	1	2	2	2	2
Read Port GP Reg. File	3	6	8	10	12
Write Port GP Reg. File	1	2	3	4	5
Boolean Unit	1	1	1	1	1
Speculative Execution ?	N/A	Yes	Yes	Yes	Yes

Table 6.1 HARP RISC dymanic cycle counts
Before and after filling branch delay slots

Program	Delay Slots Unfilled	Delay Slots Filled	Speedup
Bubblesort	623,643	601,750	1.04
Quick	286,649	271,469	1.06
Treesort	472,531	428,951	1.10
Queens	521,790	480,504	1.09
Perm	758,804	669,362	1.13
Tower	580,070	524,836	1.11
Puzzle	80,080	77,170	1.04
Intmm	242,767	236,225	1.03
Speedup (harmonic mean)			1.07

Table 6.2 Speedup over HARP RISC
Issue rate: 2

Program	Serial	Parallel	Speedup
Bubblesort	601,750	403,868	1.49
Quick	271,469	191,396	1.42
Treesort	428,951	317,710	1.35
Queens	480,504	312,682	1.54
Perm	669,362	519,402	1.29
Tower	524,836	389,380	1.35
Puzzle	77,170	51,088	1.51
Intmm	236,225	131,206	1.80
Speedup (harmonic mean)			1.45

Table 6.3 Speedup over HARP RISC
Issue rate: 3

Program	Serial	Parallel	Speedup
Bubblesort	601,750	318,260	1.89
Quick	271,469	169,672	1.60
Treesort	428,951	283,342	1.51
Queens	480,504	284,278	1.69
Perm	669,362	502,604	1.33
Tower	524,836	336,288	1.56
Puzzle	77,170	44,568	1.73
Intmm	236,225	107,094	2.21
Speedup (harmonic mean)			1.66

Table 6.4 Speedup over HARP RISC
Issue rate: 4

Program	Serial	Parallel	Speedup
Bubblesort	601,750	315,860	1.91
Quick	271,469	156,870	1.73
Treesort	428,951	278,340	1.54
Queens	480,504	276,720	1.74
Perm	669,362	502,062	1.33
Tower	524,836	325,992	1.61
Puzzle	77,170	39,280	1.96
Intmm	236,225	93,312	2.53
Speedup (harmonic mean)			1.74

Table 6.5 Speedup over HARP RISC
Issue rate: 5

Program	Serial	Parallel	Speedup
Bubblesort	601,750	314,258	1.91
Quick	271,469	152,868	1.78
Treesort	428,951	274,338	1.56
Queens	480,504	269,378	1.78
Perm	669,362	502,058	1.33
Tower	524,836	325,970	1.61
Puzzle	77,170	38,972	1.98
Intmm	236,225	87,650	2.70
Speedup (harmonic mean)			1.76

Table 6.6 Speedup over HARP RISC using two cache memory ports
Issue rate: 2

Program	Serial	Parallel	Speedup
Bubblesort	601,750	364,060	1.65
Quick	271,469	188,734	1.44
Treesort	428,951	316,978	1.35
Queens	480,504	296,660	1.62
Perm	669,362	487,598	1.37
Tower	524,836	364,964	1.44
Puzzle	77,170	51,008	1.51
Intmm	236,225	132,640	1.78
Speedup (harmonic mean)			1.51

Table 6.7 Speedup over HARP RISC using two cache memory ports
Issue rate: 3

Program	Serial	Parallel	Speedup
Bubblesort	601,750	277,254	2.17
Quick	271,469	161,648	1.68
Treesort	428,951	272,282	1.58
Queens	480,504	266,454	1.80
Perm	669,362	415,462	1.61
Tower	524,836	305,540	1.72
Puzzle	77,170	44,448	1.74
Intmm	236,225	106,294	2.22
Speedup (harmonic mean)			1.79

Table 6.8 Speedup over HARP RISC using two cache memory ports
Issue rate: 4

Program	Serial	Parallel	Speedup
Bubblesort	601,750	275,252	2.19
Quick	271,469	151,616	1.79
Treesort	428,951	265,272	1.62
Queens	480,504	245,216	1.96
Perm	669,362	395,304	1.69
Tower	524,836	289,102	1.82
Puzzle	77,170	39,140	1.97
Intmm	236,225	92,912	2.54
Speedup (harmonic mean)			1.91

Table 6.9 Speedup over HARP RISC using two cache memory ports
Issue rate: 5

Program	Serial	Parallel	Speedup
Bubblesort	601,750	273,650	2.20
Quick	271,469	147,612	1.84
Treesort	428,951	260,270	1.65
Queens	480,504	235,774	2.04
Perm	669,362	375,140	1.78
Tower	524,836	289,080	1.82
Puzzle	77,170	38,808	1.99
Intmm	236,225	87,450	2.70
Speedup (harmonic mean)			1.96

Table 6.10 Speedup over HARP RISC using two register writeback ports
Issue rate: 4

Program	Serial	Parallel	Speedup
Bubblesort	601,750	356,060	1.69
Quick	271,469	161,268	1.68
Treesort	428,951	281,356	1.52
Queens	480,504	286,180	1.68
Perm	669,362	502,072	1.33
Tower	524,836	342,458	1.53
Puzzle	77,170	43,002	1.79
Intmm	236,225	95,714	2.47
Speedup (harmonic mean)			1.66

Table 6.11 Speedup over HARP RISC using three register writeback ports
Issue rate: 4

Program	Serial	Parallel	Speedup
Bubblesort	601,750	315,860	1.91
Quick	271,469	157,756	1.72
Treesort	428,951	278,340	1.54
Queens	480,504	276,940	1.74
Perm	669,362	502,062	1.33
Tower	524,836	330,110	1.59
Puzzle	77,170	39,686	1.94
Intmm	236,225	93,272	2.53
Speedup (harmonic mean)			1.73

Table 6.12 Speedup over HARP RISC using four register writeback ports
Issue rate: 4

Program	Serial	Parallel	Speedup
Bubblesort	601,750	315,860	1.91
Quick	271,469	156,870	1.73
Treesort	428,951	278,340	1.54
Queens	480,504	276,720	1.74
Perm	669,362	502,062	1.33
Tower	524,836	325,992	1.61
Puzzle	77,170	39,280	1.96
Intmm	236,225	93,312	2.53
Speedup (harmonic mean)			1.74

Table 6.13 Speedup over HARP RISC using one and two branch units
Issue rate: 4

Program	One Branch Unit	Two Branch Units	Difference
Bubblesort	315,864	315,860	4
Quick	156,876	156,870	6
Treesort	279,344	278,340	1,004
Queens	280,740	276,720	4,020
Perm	502,064	502,062	2
Tower	325,992	325,992	0
Puzzle	39,334	39,280	54
Intmm	93,352	93,312	40
Difference (harmonic mean)			0.73

Table 14 Serial Execution Time (Cycles)

Program	ORed Indexing	Trad. Modes	Register Indirect	SP Address Adder
Bubblesort	48946	44552	48962	48946
Intmm	235624	236232	238084	235624
Perm	16032	16940	19346	16046
Puzzle	75076	77492	75316	75076
Queens	47654	49858	52174	47648
Quick	48632	51242	49716	48632
Tower	33106	34610	37798	33106
Treesort	72790	78894	74806	72790
Average	72233	73728	74525	72234
H. mean	43043	44505	47422	43055

Table 15 Parallel Execution Time (Cycles)

Program	ORed Indexing	Trad. Modes	Register Indirect	SP Address Adder
Bubblesort	26510	29562	26510	26510
Intmm	95570	96578	95570	95570
Perm	12010	13316	12010	12010
Puzzle	39302	41816	39302	39302
Queens	29278	31290	29502	29274
Quick	28698	31382	28876	28698
Tower	22952	25804	22952	22952
Treesort	47818	54198	47818	47818
Average	37767	40493	37818	37767
H. mean	27551	30278	27596	27550

Table 16 Parallel Instruction Count (Bytes)

Program	ORed Indexing	Trad. Modes	Register Indirect	SP Address Adder
Bubblesort	25871	19505	25879	25871
Intmm	37709	35709	38939	37709
Permute	7994	7994	9644	7994
Puzzle	48877	46825	48997	48997
Queens	37251	37695	39189	37248
Quick	18704	18704	19346	18704
Tower	17873	17189	19156	17873
Treesort	31733	31631	32741	31733
Average	28252	26907	29236	28255
H. mean	21136	20227	22964	21139

Table 6.17 Impact of instruction combining on sequential code

Program	Without Combining	With Combining	Speedup
Bubblesort	624,039	623,643	1.00
Quick	303,531	286,649	1.06
Treesort	475,523	472,531	1.01
Queens	524,810	521,790	1.01
Perm	786,214	758,804	1.04
Tower	580,106	580,070	1.00
Puzzle	85,768	80,080	1.07
Intmm	245,367	242,767	1.01
Speedup (harmonic mean)			1.02

Table 6.18 Impact of instruction combining on parallel code
Issue rate: 4

Program	Without Combining	With Combining	Speedup
Bubblesort	356,058	315,860	1.13
Quick	167,698	156,870	1.07
Treesort	282,352	278,340	1.01
Queens	276,860	276,720	1.00
Perm	520,326	502,062	1.04
Tower	325,992	325,992	1.00
Puzzle	40,278	39,280	1.03
Intmm	93,312	93,312	1.00
Speedup (harmonic mean)			1.03

Table 6.19 Speedup over HARP RISC using both register renaming and conditional execution for one memory port
Issue rate: 2

Program	Serial	Parallel	Speedup
Bubblesort	601,750	403,868	1.49
Quick	271,469	191,396	1.42
Treesort	428,951	317,710	1.35
Queens	480,504	312,682	1.54
Perm	669,362	519,402	1.29
Tower	524,836	389,380	1.35
Puzzle	77,170	51,088	1.51
Intmm	236,225	131,206	1.80
Speedup (harmonic mean)			1.45

Table 6.20 Speedup over HARP RISC using both register renaming and conditional execution for one memory port
Issue rate: 3

Program	Serial	Parallel	Speedup
Bubblesort	601,750	318,260	1.89
Quick	271,469	169,672	1.60
Treesort	428,951	283,342	1.51
Queens	480,504	284,278	1.69
Perm	669,362	502,604	1.33
Tower	524,836	336,288	1.56
Puzzle	77,170	44,568	1.73
Intmm	236,225	107,094	2.21
Speedup (harmonic mean)			1.66

Table 6.21 Speedup over HARP RISC using both register renaming and conditional execution for one memory port
Issue rate: 4

Program	Serial	Parallel	Speedup
Bubblesort	601,750	315,860	1.91
Quick	271,469	156,870	1.73
Treesort	428,951	278,340	1.54
Queens	480,504	276,720	1.74
Perm	669,362	502,062	1.33
Tower	524,836	325,992	1.61
Puzzle	77,170	39,280	1.96
Intmm	236,225	93,312	2.53
Speedup (harmonic mean)			1.74

Table 6.22 Speedup over HARP RISC using both register renaming and conditional execution for one memory port
Issue rate: 5

Program	Serial	Parallel	Speedup
Bubblesort	601,750	314,258	1.91
Quick	271,469	152,868	1.78
Treesort	428,951	274,338	1.56
Queens	480,504	269,378	1.78
Perm	669,362	502,058	1.33
Tower	524,836	325,970	1.61
Puzzle	77,170	38,972	1.98
Intmm	236,225	87,650	2.70
Speedup (harmonic mean)			1.76

Table 6.23 Speedup over HARP RISC using conditional execution only for one memory port
Issue rate: 2

Program	Serial	Parallel	Speedup
Bubblesort	601,750	363,680	1.65
Quick	271,469	189,190	1.43
Treesort	428,951	327,580	1.31
Queens	480,504	332,242	1.45
Perm	669,362	519,406	1.29
Tower	524,836	412,140	1.27
Puzzle	77,170	48,340	1.60
Intmm	236,225	133,264	1.77
Speedup (harmonic mean)			1.45

Table 6.24 Speedup over HARP RISC using conditional execution only for one memory port
Issue rate: 3

Program	Serial	Parallel	Speedup
Bubblesort	601,750	318,260	1.89
Quick	271,469	175,170	1.55
Treesort	428,951	296,706	1.45
Queens	480,504	297,718	1.61
Perm	669,362	502,068	1.33
Tower	524,836	379,436	1.32
Puzzle	77,170	45,124	1.71
Intmm	236,225	109,174	2.16
Speedup (harmonic mean)			1.59

Table 6.25 Speedup over HARP RISC using conditional execution only for one memory port
Issue rate: 4

Program	Serial	Parallel	Speedup
Bubblesort	601,750	316,258	1.90
Quick	271,469	166,658	1.63
Treesort	428,951	290,706	1.48
Queens	480,504	294,960	1.63
Perm	669,362	502,070	1.33
Tower	524,836	369,158	1.42
Puzzle	77,170	41,716	1.85
Intmm	236,225	95,610	2.47
Speedup (harmonic mean)			1.66

Table 6.26 Speedup over HARP RISC using conditional execution only for one memory port
Issue rate: 5

Program	Serial	Parallel	Speedup
Bubblesort	601,750	314,656	1.91
Quick	271,469	162,656	1.67
Treesort	428,951	286,704	1.50
Queens	480,504	295,438	1.63
Perm	669,362	502,062	1.33
Tower	524,836	369,154	1.42
Puzzle	77,170	41,586	1.86
Intmm	236,225	90,148	2.62
Speedup (harmonic mean)			1.67

Table 6.27 Speedup over HARP RISC using register renaming only for one memory port
Issue rate: 2

Program	Serial	Parallel	Speedup
Bubblesort	601,750	442,418	1.36
Quick	271,469	195,080	1.39
Treesort	428,951	332,862	1.29
Queens	480,504	366,502	1.31
Perm	669,362	529,490	1.26
Tower	524,836	397,650	1.32
Puzzle	77,170	53,688	1.44
Intmm	236,225	133,898	1.76
Speedup (harmonic mean)			1.38

Table 6.28 Speedup over HARP RISC using register renaming only for one memory port
Issue rate: 3

Program	Serial	Parallel	Speedup
Bubblesort	601,750	356,814	1.69
Quick	271,469	176,262	1.54
Treesort	428,951	299,728	1.43
Queens	480,504	339,258	1.42
Perm	669,362	509,314	1.31
Tower	524,836	365,100	1.44
Puzzle	77,170	47,272	1.63
Intmm	236,225	109,412	2.16
Speedup (harmonic mean)			1.54

Table 6.29 Speedup over HARP RISC using register renaming only for one memory port
Issue rate: 4

Program	Serial	Parallel	Speedup
Bubblesort	601,750	354,414	1.70
Quick	271,469	166,174	1.63
Treesort	428,951	293,732	1.46
Queens	480,504	324,980	1.48
Perm	669,362	509,322	1.31
Tower	524,836	358,920	1.46
Puzzle	77,170	47,146	1.64
Intmm	236,225	95,830	2.47
Speedup (harmonic mean)			1.59

Table 6.30 Speedup over HARP RISC using register renaming only for one memory port
Issue rate: 5

Program	Serial	Parallel	Speedup
Bubblesort	601,750	352,812	1.71
Quick	271,469	159,810	1.70
Treesort	428,951	289,730	1.48
Queens	480,504	324,978	1.48
Perm	669,362	509,314	1.31
Tower	524,836	358,916	1.46
Puzzle	77,170	44,450	1.74
Intmm	236,225	90,368	2.61
Speedup (harmonic mean)			1.62

Table 6.31 Speedup over HARP RISC using both register renaming and conditional execution for two memory ports
Issue rate: 2

Program	Serial	Parallel	Speedup
Bubblesort	601,750	364,060	1.65
Quick	271,469	188,734	1.44
Treesort	428,951	316,978	1.35
Queens	480,504	296,660	1.62
Perm	669,362	487,598	1.37
Tower	524,836	364,964	1.44
Puzzle	77,170	51,008	1.51
Intmm	236,225	132,640	1.78
Speedup (harmonic mean)			1.51

Table 6.32 Speedup over HARP RISC using both register renaming and conditional execution for two memory ports
Issue rate: 3

Program	Serial	Parallel	Speedup
Bubblesort	601,750	277,254	2.17
Quick	271,469	161,648	1.68
Treesort	428,951	272,282	1.58
Queens	480,504	266,454	1.80
Perm	669,362	415,462	1.61
Tower	524,836	305,540	1.72
Puzzle	77,170	44,448	1.74
Intmm	236,225	106,294	2.22
Speedup (harmonic mean)			1.79

Table 6.33 Speedup over HARP RISC using both register renaming and conditional execution for two memory ports
Issue rate: 4

Program	Serial	Parallel	Speedup
Bubblesort	601,750	275,252	2.19
Quick	271,469	151,616	1.79
Treesort	428,951	265,272	1.62
Queens	480,504	245,216	1.96
Perm	669,362	395,304	1.69
Tower	524,836	289,102	1.82
Puzzle	77,170	39,140	1.97
Intmm	236,225	92,912	2.54
Speedup (harmonic mean)			1.91

Table 6.34 Speedup over HARP RISC using both register renaming and conditional execution for two memory ports
Issue rate: 5

Program	Serial	Parallel	Speedup
Bubblesort	601,750	273,650	2.20
Quick	271,469	147,612	1.84
Treesort	428,951	260,270	1.65
Queens	480,504	235,774	2.04
Perm	669,362	375,140	1.78
Tower	524,836	289,080	1.82
Puzzle	77,170	38,808	1.99
Intmm	236,225	87,450	2.70
Speedup (harmonic mean)			1.96

Table 6.35 Speedup over HARP RISC using conditional execution only for two memory ports
Issue rate: 2

Program	Serial	Parallel	Speedup
Bubblesort	601,750	363,672	1.65
Quick	271,469	187,410	1.45
Treesort	428,951	323,074	1.33
Queens	480,504	316,400	1.52
Perm	669,362	470,278	1.42
Tower	524,836	366,992	1.43
Puzzle	77,170	48,180	1.60
Intmm	236,225	132,440	1.78
Speedup (harmonic mean)			1.51

Table 6.36 Speedup over HARP RISC using conditional execution only for two memory ports
Issue rate: 3

Program	Serial	Parallel	Speedup
Bubblesort	601,750	317,054	1.90
Quick	271,469	167,146	1.62
Treesort	428,951	287,704	1.49
Queens	480,504	272,834	1.76
Perm	669,362	432,784	1.55
Tower	524,836	334,290	1.57
Puzzle	77,170	44,962	1.72
Intmm	236,225	107,542	2.20
Speedup (harmonic mean)			1.70

Table 6.37 Speedup over HARP RISC using conditional execution only for two memory ports
Issue rate: 4

Program	Serial	Parallel	Speedup
Bubblesort	601,750	315,450	1.91
Quick	271,469	158,744	1.71
Treesort	428,951	280,700	1.53
Queens	480,504	266,036	1.81
Perm	669,362	412,630	1.62
Tower	524,836	328,148	1.60
Puzzle	77,170	41,556	1.86
Intmm	236,225	93,978	2.51
Speedup (harmonic mean)			1.78

Table 6.38 Speedup over HARP RISC using conditional execution only for two memory ports
Issue rate: 5

Program	Serial	Parallel	Speedup
Bubblesort	601,750	313,848	1.92
Quick	271,469	154,742	1.75
Treesort	428,951	274,700	1.56
Queens	480,504	266,034	1.81
Perm	669,362	392,466	1.71
Tower	524,836	324,010	1.62
Puzzle	77,170	41,424	1.86
Intmm	236,225	88,316	2.67
Speedup (harmonic mean)			1.82

Table 6.39 Speedup over HARP RISC using register renaming only for two memory ports
Issue rate: 2

Program	Serial	Parallel	Speedup
Bubblesort	601,750	402,614	1.49
Quick	271,469	193,302	1.40
Treesort	428,951	330,858	1.30
Queens	480,504	355,200	1.35
Perm	669,362	477,528	1.40
Tower	524,836	368,924	1.42
Puzzle	77,170	53,608	1.44
Intmm	236,225	133,282	1.77
Speedup (harmonic mean)			1.44

Table 6.40 Speedup over HARP RISC using register renaming only for two memory ports
Issue rate: 3

Program	Serial	Parallel	Speedup
Bubblesort	601,750	356,008	1.69
Quick	271,469	172,484	1.57
Treesort	428,951	292,728	1.47
Queens	480,504	327,956	1.47
Perm	669,362	437,196	1.53
Tower	524,836	332,258	1.58
Puzzle	77,170	47,152	1.64
Intmm	236,225	107,996	2.19
Speedup (harmonic mean)			1.62

Table 6.41 Speedup over HARP RISC using register renaming only for two memory ports
Issue rate: 4

Program	Serial	Parallel	Speedup
Bubblesort	601,750	354,010	1.70
Quick	271,469	163,396	1.66
Treesort	428,951	286,732	1.50
Queens	480,504	318,478	1.51
Perm	669,362	437,204	1.53
Tower	524,836	330,212	1.59
Puzzle	77,170	44,658	1.73
Intmm	236,225	94,814	2.49
Speedup (harmonic mean)			1.67

Table 6.42 Speedup over HARP RISC using register renaming only for two memory ports
Issue rate: 5

Program	Serial	Parallel	Speedup
Bubblesort	601,750	352,406	1.71
Quick	271,469	157,032	1.73
Treesort	428,951	281,730	1.52
Queens	480,504	313,676	1.53
Perm	669,362	437,196	1.53
Tower	524,836	326,074	1.61
Puzzle	77,170	44,330	1.74
Intmm	236,225	89,352	2.64
Speedup (harmonic mean)			1.70