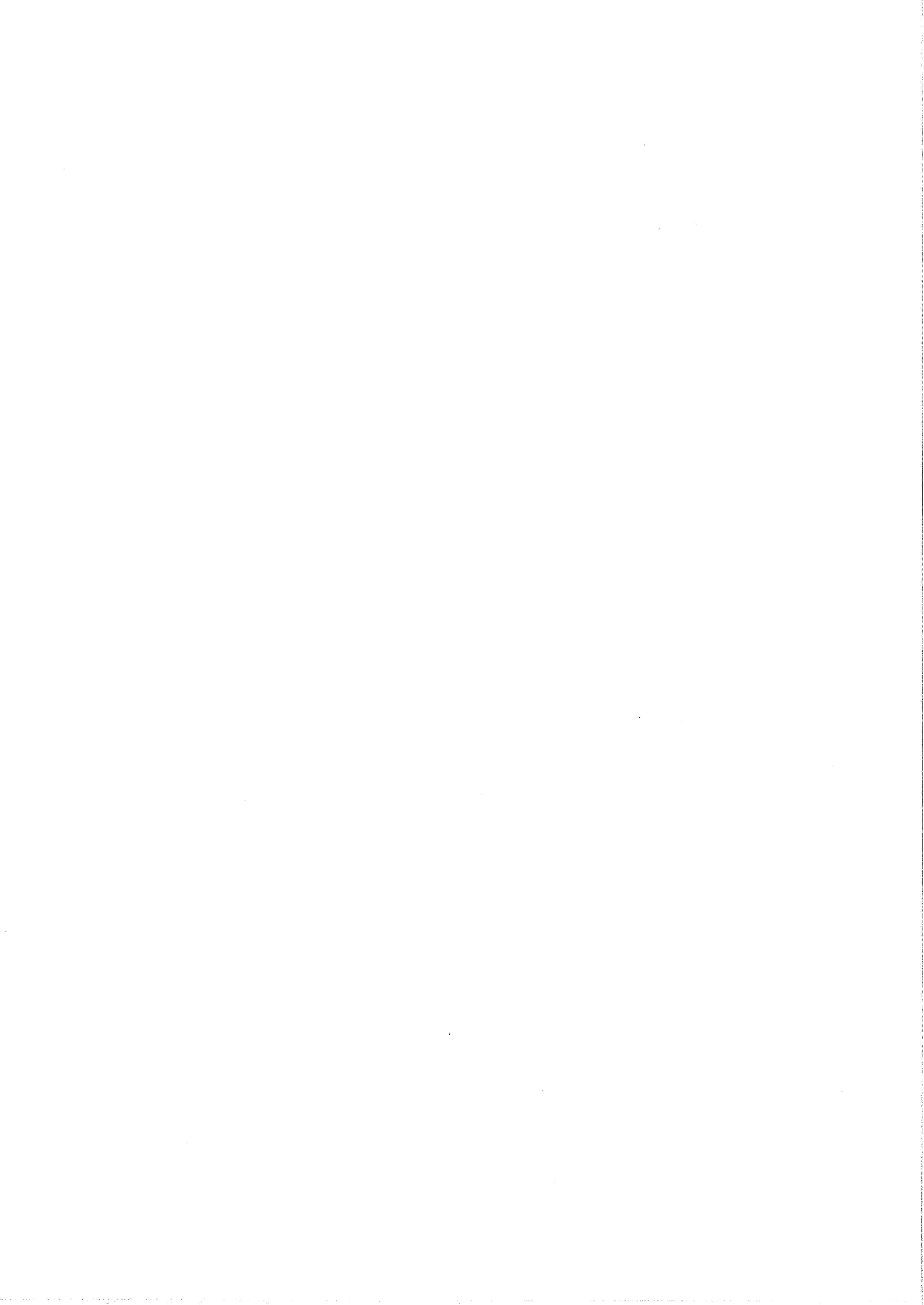
DIVISION OF COMPUTER SCIENCE

Using a Resource Limited Instruction Scheduler to Evaluate the iHARP Processor

F L Steven
G B Steven
L Wang

Technical Report No.198

May 1994



Using a Resource Limited Instruction Scheduler to Evaluate the iHARP Processor

Abstract

RISC processors have approached an execution rate of one instruction per cycle by using pipelining to speed up execution. However, to achieve an execution rate of more than one instruction per cycle, processors must issue multiple instructions in each processor cycle. This paper evaluates the architectural features of iHARP, a VLIW (Very Long Instruction Word) processor with an instruction issue rate of four, which has been developed at the University of Hertfordshire. A distinctive feature of iHARP is the provision of Boolean guards on all instructions. Instructions are then only executed at run time if the attached Boolean guard is true. A second distinctive feature is the use of a simplified addressing ORed indexing mechanism to avoid load delays. This paper evaluates the benefits of both these features and quantifies their performance advantage. Other architectural features evaluated include instruction issue rate, code size, number of data cache ports, number of register file write ports, number of branch units, instruction combining and loop unrolling. The evaluation uses RLS, a resource limited instruction scheduler, specifically developed to statically reorder code for parallel execution on iHARP.

Key Words

VLIW Superscalar Instruction Scheduling Guarded Instruction Execution

1. INTRODUCTION

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

This paper describes the development of RLS, a resource limited instruction scheduling system, and its use in the evaluation of the iHARP architecture [2]. The features considered include instruction issue rate, code size, number of data cache ports, addressing mechanisms, number of register file write ports and number of branch units. The usefulness of the HARP guarded execution facility, which allows a Boolean guard to be attached to every iHARP instruction, is also quantified.

The remainder of this paper is organised as follows: Sections 2 and 3 discuss the fundamental limitations on instruction-level parallelism faced by all MII processors; section 4 reviews other research in the area of static instruction scheduling; section 5 describes the compiler/scheduler software and the HARP models used in this research; section 6 contains our evaluation of iHARP and section 7 offers some concluding remarks.

2. DATA DEPENDENCIES

Scheduling instructions for parallel execution can be viewed as a process in which each instruction is successively moved or percolated [3] up through the code structure in an attempt to ensure that it is executed at the earliest possible opportunity. Ultimately, further code motion is blocked when a data dependency on an already scheduled instruction is encountered.

Three classes of data dependencies can be identified: Read-after-write (RAW), write-after-read (WAR) and write-after-write (WAW). WAR data dependencies are also known as anti-dependencies while WAW dependencies are also called output dependencies. In the following code fragment a read-after-write data dependency exists between the two instructions:

```
ADD R1, R2, R3
ADD R4, R1, R5
```

Since no instruction scheduler can safely reverse the order of these instructions, RAW data dependencies are true data dependencies which ultimately limit the performance of all MII processors.

2.1 Register Renaming

In contrast, WAR and WAW data dependencies can both be removed using register renaming. For example, in the fragment below the second instruction has an anti-dependence on the first instruction.

```
ADD R5, R6, R7
SUB R6, R8, #256
```

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

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

The MOV instruction is required to restore the new result to R6. This additional instruction need not necessarily introduce further data dependencies since subsequent instructions using R6 can equally well use R20.

2.2 Speculative Execution

An instruction is executed speculatively if it is executed before it is known whether the path containing the instruction will actually be taken. Consider the following example:

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

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

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

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

However, if R6 is live immediately after the branch instruction, its value will have been corrupted by the code motion. Register renaming can be used to avoid this problem:

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

Label:  MOV R6, R20
```

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

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

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

Label:
```

Now the Boolean guard ensures that the LD will only be executed if the branch is taken. However, further percolation of the LD instruction will move it beyond the scope of the Boolean guard. Now only register renaming can be used to avoid corrupting R6:

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

Label:  MOV R6, R20
```

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

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

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

Now assume that both the load and subtract instructions are scheduled speculatively ahead of the branch. For this code motion to be safe both R6 and R8 must be dead at Label.

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

If the load instruction generates an exception, R6 will be marked as polluted. Since the subtract instruction is also executed speculatively, it will in turn mark R8 as polluted when it attempts to use the polluted value in R6. An exception will only be taken when the non-speculative relational instruction attempts to use the polluted value held in R8. Note this is the earliest point in the code where it is certain that the speculative load should be executed. In contrast, if the branch is taken no exception will be taken.

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

2.3 Memory Disambiguation

Data dependencies may also involve memory locations. Memory disambiguation is used to determine whether two instructions access the same memory location and are therefore dependent. Consider the following code fragment:

```
ST 24(SP), R6
LD R4, 8(SP)
```

Clearly, the memory locations referred to in the ST and LD instructions are different and the LD can be safely moved ahead of the ST. However, suppose the code is as follows:

```
ST 8(SP), R6
LD R4, 8(R5)
```

Now it must be shown that registers SP and R5 do not hold the same address before the LD can be safely percolated ahead of the ST.

An instruction scheduler for a high-performance multiple-instruction-issue processor must therefore include a module which attempts to disambiguate pairs of memory references. Successful disambiguation then allows the scheduler to percolate loads ahead of stores. However, to ensure correct program execution, the scheduler must always assume that two references refer to the same memory location unless it can be shown otherwise.

3. AVAILABLE INSTRUCTION LEVEL PARALLELISM

A number of groups have attempted to quantify the amount of fine-grained parallelism available in general-purpose code. Wall [4] used simulations based on instruction traces to investigate the parallelism available to superscalar processors. Even with perfect renaming and memory disambiguation, the parallelism realised rarely exceeded seven and was typically only five. However, when Wall substituted perfect branch prediction for his hardware branch prediction model, the amount of parallelism realised increased spectacularly. Yale Patt's [5] group also used trace driven simulations to investigate superscalar performance. The group concluded that current technology could achieve execution rates of between two and six instructions per cycle and looked forward to significantly faster execution rates in the future. By far the most spectacular upper bounds on fine-grained parallelism were reported by Lam [6]. Lam recognised that superscalar processors, which rely solely on hardware to exploit parallelism, can only extract parallelism between successive branch mispredictions. This follows because after a branch prediction fails any results which have been generated by the speculative execution of instructions following the branch must first be discarded before execution is restarted at the correct branch successor instruction. Avoiding this restriction significantly increases the available parallelism. As a consequence, Lam's results range from two instructions per cycle for her basic model to an average of no fewer than 159 instructions per cycle for an ORACLE model with perfect branch prediction.

All of the above studies emphasise that significantly more parallelism is available to MII processors than is realised by current designs. This gap reflects both limited hardware resources and the current early stage of development of instruction scheduling technology. The role of accurate branch prediction is also emphasised, reflecting the inability of current

superscalars to extract parallelism across mispredicted branches. Significantly the static instruction scheduling approach used in iHARP removes this dependence on branch prediction by percolating instructions across branches from both successor basic blocks. Parallelism is then limited only by true data dependencies, unresolved memory ambiguities and, of course, by resources and the instruction scheduling technology.

4. INSTRUCTION SCHEDULING

This section reviews current work on instruction scheduling. In general, global instruction scheduling can be divided into low-level and high-level components. Low-level code scheduling is concerned with packing individual instructions into long instruction words for parallel execution at run time. High-level scheduling involves scheduling precedence, transformations of the program graph and applications of low-level scheduling.

Low-level scheduling can follow two approaches. In the first approach each long instruction word is filled in turn from a list of candidate instructions. This approach is sometimes termed list scheduling [7]. Alternatively, individual instructions can be percolated upwards through a data structure representing the current state of the partially scheduled program. This approach builds on the pioneering work of Nicolau [3] who defined a set of primitives which allow individual instructions to be safely moved, one step at a time, through a program graph.

Whichever approach is followed, the full benefits of instruction scheduling can only be realised if code is scheduled across multiple basic blocks. For example, in Trace Scheduling [8] the scheduler attempts to identify likely paths or traces through the code and then schedules each trace as if it were a single basic block. Compensation code is then added to off-trace paths to preserve the program semantics. As a result the execution time of traces which are selected for early scheduling is optimised at the expense of less-important traces which are scheduled later. Similarly, the IMPACT group at the University of Illinois enlarges the scope of its instruction schedulers by combining basic blocks into superblocks [9] and hyperblocks [10].

Ideally the scope for code motion should be a whole procedure or even a whole program. Enlarging basic blocks to form traces or superblocks still leaves barriers to code motion between these larger units. For example, in Trace Scheduling, since a trace cannot traverse a loop back edge, no attempt is made to overlap code between successive iterations of a loop body. Instead aggressive loop unrolling is used to extend the size of the loop body. Inevitably, this unrolling results in excessive code size.

Software pipelining [11] is a method for overlapping the operations from successive loop iterations without unrolling the loop. For example, consider the following code:

```
for i = 0 to n
        a;
        b;
        c;
    end for;
```

Using loop unrolling, successive loop iterations might be overlapped as follows:

a_0
$a_1\ b_0$
$a_2\ b_1\ c_0$
$a_3\ b_2\ c_1$
.. .. ..
$a_n\ b_{n-1}\ c_{n-2}$
$\quad b_n\ c_{n-1}$
$\qquad c_n$

Note that after the second line, a steady state is reached in which the same three instructions are executed in each cycle. In software pipelining this steady state is identified and transformed into a loop. For simplicity the loop iteration controls are omitted.

a_0
$a_1\ b_0$
loop: $a_{i+2}\ b_{i+1}\ c_i$ bcc loop ; where 0 <= i <= n - 2
$b_n\ c_{n-1}$
c_n

Software pipelining therefore attempts to achieve the benefits of complete loop unrolling by replacing an unrolled loop by a prologue, a steady-state loop body and an epilogue. In the above example, during each iteration of the scheduled loop body, instructions from three successive iterations of the original loop are always being executed in parallel. The scheduling problem in software pipelining is to obtain a steady-state loop body which minimises the initiation interval between successive loop iterations. This task becomes increasingly difficult as the complexity of the loop is increased.

Enhanced Percolation Scheduling [12] is a particularly elegant method of achieving software pipelining which has been developed by Ebcioglu's team at IBM. Enhanced Percolation Scheduling uses improved versions of Nicolau's percolation primitives to schedule code within a loop and to move code across loop back edges. This additional code motion automatically realises software pipelining with loops of arbitrary complexity.

At IBM, Enhanced Percolation Scheduling is applied to an unconventional VLIW architecture based on tree instructions. Excessive processor resources are also postulated, and it is assumed that all branches can be resolved before the next instruction fetch is initiated. The RLS scheduler can be viewed as a first step towards applying Enhanced Percolation Scheduling to a more conventional processor architecture where resources are limited and where branch instructions are only resolved after the immediately following VLIW instruction has been fetched from the instruction cache.

5. THE HARP PROJECT

The aim of the HARP project was to develop an MII architecture which could sustain an instruction execution rate significantly in excess of one instruction per cycle. As part of the project iHARP [13], a VLIW processor with an instruction issue rate of four, was designed, fabricated and tested.

5.1 iHARP Processor

The iHARP processor provides 32, general-purpose registers and eight, 1-bit boolean registers. The four parallel pipelines share the general-purpose register file. iHARP uses a four-stage pipeline. In the IF stage four instructions are fetched from the instruction cache; in the RF stage register operands are accessed; in the ALU/MEM stage instructions are executed or access the data cache; finally in the WB stage results are returned to the register file. To allow the data cache to be accessed in the third pipeline stage, all addresses are formed in the RF stage. The simplified ORed indexing addressing mechanism used on iHARP makes this possible without extending the processor cycle time. Complete operand bypassing then allows the results of ALU and load instructions to be used in the following cycle. Since branch conditions are also resolved in the RF stage, the branch delay is one.

While all pipelines are able to support ALU, relational and a limited range of shift instructions, the functionality of each pipeline is restricted by the need to conserve hardware resources (Fig1). As a result only two pipelines, one and three, support branch instructions. Similarly memory reference instructions are restricted to pipelines zero and two. However, while two branches can be executed in parallel, the single data cache port precludes the parallel execution of two memory references instructions. Finally 32-bit literals are introduced by providing 64-bit instructions which occupy two adjacent instruction positions.

5.2 MII HARP Architectures

To evaluate the HARP architecture, a family of MII architectures with different instruction issue rates was postulated. Each model has the same instruction set, addressing modes and

pipeline stages as iHARP and provides hardware support for speculative instruction execution. It is assumed that every pipeline will support ALU, relational and shift operations. Also, in line with iHARP, a maximum of two branch instructions and one memory reference instruction can be issued in each cycle. These base parameters were then systematically varied to evaluate the architecture.

5.3 HARP Compiler and Instruction Scheduler

A HARP C compiler was generated using the GNU CC compiler generator [14]. The sequential HARP code produced by the compiler was then scheduled for multiple instruction issue using a Resource Limited Scheduler (RLS) developed by Liang Wang [2]. The HARP simulator [15] was used to execute both sequential and parallel code.

RLS is a resource limited scheduler which was developed specifically to exploit fine-grained parallelism in iHARP. Since iHARP has only four pipelines, clearly the scheduler has only limited resources at its disposal. RLS also aims to control the code expansion usually associated with VLIW architectures by ensuring that the number of long instructions generated never exceeds the original number of sequential instructions. The scheduler is also 'resource limited' in this sense.

Conceptually RLS (Fig2) consists of a high level and a low level. The high level transforms the sequential instructions of a procedure into a linked data structure, detects basic blocks and constructs a flow graph. It then uses a set of heuristics to select the next basic block for the low-level scheduler. The low-level scheduler percolates individual instructions from a basic block into an instruction graph of previously scheduled instructions. Guarded instruction execution, renaming and memory disambiguation are all used to increase parallelism. Also after each loop body has been scheduled, an attempt is made to move code across the loop back edge to reduce the size of the loop and to implement software pipelining.

Finally after each procedure has been scheduled, a separate post-pass performs inter-procedural instruction scheduling. During this phase RLS attempts to percolate the initial LIW instructions from each procedure into the body of the calling routine. Only entire LIW instructions are moved to allows RLS to preserve the program semantics by simply adjusting the procedure entry points.

RLS is a research scheduler and as such no attempt was made to optimise the time taken to schedule the benchmarks. Interestingly, however, the scheduling time was very similar to the compilation time.

6. iHARP EVALUATION

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

6.1 Instruction Issue Rate

Fig3 shows the average speedups achieved by RLS as a function of instruction issue rate. Each model is assumed to have an ALU per pipeline, two branch units and a single data cache port. The average speedups obtained for issue rates of two, three, four, five and eight are 1.45, 1.66, 1.74, 1.76 and 1.77 respectively. RLS therefore performs well for iHARP, its four-pipeline target processor, but fails to deliver significant additional parallelism as further pipelines are added.

This performance gain was only achieved at the cost of significant code expansion. Code size increased by 1.38, 1.86, 2.34 and 2.82 for issue rates of two, three, four and five (Fig4). Nonetheless, RLS comfortably achieves its design target of ensuring that the number of long instruction words after scheduling never exceeds the initial count of sequential instructions.

One of the main disadvantages of a VLIW processor is the disproportionate number of NOPs introduced into the code. It is therefore instructive to compare HARP with a minimal superscalar architecture executing scheduled HARP code from which all the NOPs have been removed. Even with in-order instruction issue such a superscalar would, in the worst case, simply reconstruct the long instruction word schedule generated for HARP. Performance would therefore equal or exceed HARP at all issue rates. However, since all the NOPs have been removed, code size would be dramatically reduced to the number of operations in the HARP code. Code expansion is then only 15% for an issue rate of two, rising to 18% for an issue rate of five (Fig4).

6.2 Number of Cache Memory Ports

Some of the benchmarks execute a high percentage of memory reference instructions. For these programs a single data cache port represents an obvious performance bottleneck, particularly as the instruction issue rate is increased. In spite of the high cost, it is therefore useful to consider adding additional data cache ports. With two data cache ports, the

average performance of a four pipeline processor is improved by approximately 10%. The speedups now range from 1.51 with an issue rate of two rising to 1.97 with an issue rate of eight (Fig5). With three cache ports, the average performance of a four pipeline processor is improved by approximately 14%. Speedups here range from 1.84 with an issue rate of three rising to 2.04 with an issue rate of eight (Fig5).

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

6.3 Interprocedural Scheduling

The speedups shown in sections 6.1 and 6.2 are further improved if the post-pass inter-procedural scheduling phase is incorporated into RLS. With one data cache port and an issue rate of four, inter-procedural scheduling improves the performance of the four pipeline HARP model by 3.5%. The average speedups obtained with one data port are 1.51, 1.80 and 1.89 for issue rates of two, four and eight respectively (Fig6).

Performance with two and three data cache ports is also improved. With an issue rate of four, interprocedural scheduling improves the performance of the two-port model by 5.8% and the three port model by 4.6%. With two data cache ports, the speedups range from 1.59 with an issue rate of two rising to 2.13 with an issue rate of eight. With three data cache ports, the speedups range from 1.92 with an issue rate of three rising to 2.21 with an issue rate of eight (Fig6).

These improvements are relatively high given the modest nature of the RLS inter-procedural scheduling algorithm and partially reflect the high proportion of recursive routines in the Stanford benchmarks. They nonetheless encourage the belief that inter-procedural scheduling will prove to be a fruitful source of low-level parallelism in the future.

The speedups obtained compare favourably with other groups working in the area. For example, Horowitz's group at Stanford has proposed the use of boosting [16] to support speculative instruction execution in MII architectures. Boosting is an architectural mechanism which uses shadow register structures to support code motion across branch edges. An instruction is said to be boosted if it is percolated across one or more branch edges. With issue rates of two, the Stanford group achieve speedups of 1.24 for their basic model. This figure increases to 1.45 with one level of boosting and to 1.5 with three levels. The comparable HARP model with an issue rate of two and one data port achieves a speedup of 1.5, without the overhead of the complex shadow register structure required to

support boosting.

At Illinois the IMPACT group [9] use superblocks to achieve the following speedups with an issue rate of four: 2.6 with no restrictions, 2.0 with one data port and finally 1.74 with one data port and one branch unit. In comparison HARP achieves a speedup of 2.17 with three data ports and numerous other scheduling restrictions. With only one cache port the HARP speedup falls to 1.8, but this figure is maintained if only one branch unit is available.

6.4 Guarded Instruction Execution

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

Fig7 and 8 demonstrate the performance benefits of guarded instruction execution. With one memory port, the speedups obtained using both register renaming and guarded execution range from 1.45 to 1.77. These figures fall to 1.45 and 1.68 if only guarded execution is used and to 1.38 and 1.63 if only renaming is used (Fig7). Similar results are recorded with two data cache ports (Fig8). With both renaming and guarded instruction execution, speedups range from 1.51 to 1.97. These figures then fall to 1.51 and 1.84 with guarded execution only and to 1.44 and 1.72 with renaming only. Since, in practice, there is little point in using guarded execution on its own, these results suggest that guarded execution will improve performance by 9.4% with one memory port and by 14.4% with two memory ports.

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

Guarded instruction execution has three main advantages: First, it avoids restoring code and therefore conserves resources; second, it avoids using additional registers for renaming; third, it allows store instructions to be percolated across conditional branch edges. Sole reliance on guarded execution also avoids live variable analysis and speculative execution. There are two main disadvantages: First, guarded execution introduces a new data

dependence between the instruction which defines the Boolean guard and the instruction being moved; second, guarded execution can not be used to remove WAR and WAW data dependencies.

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

6.5 ORed Indexing

A RISC processor typically requires five pipeline stages to execute a load instruction [19]:

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 5.1:

IF:	Instruction Fetch.
RF:	Register Fetch.
ALU/MEM:	Perform operation or access data cache.
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 computation to the RF stage is to reduce the computation to a bitwise logical OR between the two address components [20]. 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 of n bits can then be added safely to SP using a logical OR. Glew [21] suggested the term ORed indexing to describe this addressing mechanism.

To evaluate alternative addressing mechanisms for iHARP, the machine specification for the GNUCC iHARP compiler [2] was modified to generate two further compilers. Apart from the use of different addressing modes, the iHARP instruction set was used throughout. Only the second compiler, which uses an ALU stage to compute addresses, requires a load delay. The other two compilers avoid a load delay by using simplified addressing mechanisms.

The standard compiler uses the iHARP addressing mechanisms: offset(Ri) and (Ri,Rj) where the two address components are logically ORed. Ri and Rj can be any general-purpose register. Since register R0 is always zero, register indirect and direct addressing are also available. The second compiler uses traditional RISC addressing modes: offset(Ri) and (Ri,Rj) where the address components are now added. Since a five stage pipeline is now required the load delay is one. Finally, the third compiler is restricted to register indirect and direct addressing. Since neither addressing mode requires the ALU, the four stage HARP pipeline can be used and a load delay avoided.

To use ORed indexing, the code generated on procedure entry must align the stack pointer on a power of two memory address boundary. Local data can then be safely accessed relative to the stack pointer using ORed indexing. The worst case code is shown below:

```
        BSR RA, _proc              ; return address in RA

_proc:
        MOV SP', SP                ; save old stack pointer
        AND SP, SP, #mask          ; force SP to 2^n boundary
        SUB SP, SP, #frame_size    ; allocate stack frame
        ST 4(SP), SP'              ; save old SP
        ST 0(SP), RA               ; save return address
        . .
        LD RA, 0(SP)               ; load return address
        LD SP, 4(SP)               ; restore old SP
        MOV PC, RA                 ; return from procedure
```

On entry into the procedure body the value of the old SP is saved. The AND instruction then rounds down the stack pointer to the power of two boundary required by forcing the bottom n bits to zero. The new stack frame is then allocated by subtracting the frame size from SP. On exit from the procedure the old SP is restored.

In the worst case, all the above code is required. However, most iHARP procedures use a standard minimum stack frame size of 128 bytes. Since SP is always aligned on a 128 byte boundary, this removes the need to save the old stack pointer and to realign SP before allocating a new stack frame. As a result the overwhelming majority of procedure calls incur no additional overhead to support ORed indexing. The simplified code is shown below:

```
_proc:
        SUB SP, SP, #128      ; allocated stack frame
        ST 0(SP), RA          ; save return address
        . .
        LD RA, 0(SP)          ; load return address
        ADD SP, SP, #128      ; deallocate stack frame
        MOV PC, RA            ; return from procedure
```

Each compiler was used to generate sequential code for the Stanford benchmarks. Where possible, branch and load delays were filled with an average success rate of 73%. RLS was then used to generate parallel code for the four pipeline iHARP processor.

For serial code, the best execution times were obtained with ORed indexing [Fig9]. Using traditional RISC addressing modes degraded performance by 3%, while using register indirect addressing degraded performance by 10%.

However, the move from serial to parallel code significantly changed the relative performance. With parallel code, ORed indexing and register indirect addressing performed equally well, while using traditional addressing mechanisms degraded performance by 10%. In both cases the performance advantage over traditional addressing mechanisms was achieved by executing more instructions. Using ORed indexing 4% more instructions were executed, while with register indirect addressing 14% more instructions were 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.

Register indirect addressing is also used as the principal addressing mechanism on the VIPER VLIW processor [23]. It is interesting that this group found that register indirect addressing yielded a similar 8.4% performance improvement over more traditional addressing mechanisms.

6.6 Number of Register Writeback Ports

Throughout this study it has been assumed that sufficient write ports were always provided on the general-purpose register file to allow all results to be returned to a register in the final pipeline stage. This is equivalent to assuming that the number of write ports is always equal to the number of pipelines. In practice, multiple write ports can be costly to implement. The performance impact of reducing the number of write ports in a four pipeline model was therefore investigated. On average, reducing the number of write ports from four to three degraded performance by a negligible 0.6%, while reducing the number of ports to two reduced performance by only 4.6%.

The iHARP chip was implemented with only two write-back ports on the register file. This design compromise therefore reduced performance by less than 5%. An additional mechanism was provided on iHARP to enable results to be bypassed directly to the following long instruction word without being written to the register file [13]. The objective was to reduce the pressure on the register file write ports. RLS did not attempt to use this facility, since the maximum possible gain is less than 5%.

6.7 Parallel Execution of Branch Instructions

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

6.8 Combined Instructions

A further feature of iHARP is the availability of instructions which combine a shift operation with an ALU operation. Since the shift units immediately precede the ALUs, combined instructions are still executed in one cycle. For example, consider the following

two instructions.

```
ASL R6, R7, #2        ; arithmetic shift left
ADD R5, R6, R2
```

Providing R6 is not live after the ADD instruction these two instructions can be combined in a single instruction:

```
ADD R5, R7(ASL #2), R2
```

Alternatively if R6 is still live, the following two instructions can be generated:

```
ASL R6, R7, #2
ADD R5, R7(ASL #2), R2
```

While no code is saved, the two instructions can now be scheduled independently.

Combining is implemented wherever possible by the HARP GNUCC compiler, but has a very small impact on execution time. Overall performance is improved by 2% with serial code and 3% with an instruction issue rate of four. In general, the HARP instruction set therefore allows too few instruction pairs to be combined to make a significant impact.

6.9 Loop Unrolling

Since RLS achieves software pipelining by percolating instructions across loop back edges, loop unrolling has never played a major role in RLS scheduling stategy. Nevertheless since HARP has a branch delay of one, no unrolled loop can ever start a new loop iteration more frequently than once every two cycles. Therefore if a loop iteration interval of one is ever to be achieved some loop unrolling is essential.

With these observations in mind, RLS unrolls simple loops, consisting of no more than two basic blocks. This unrolling is performed only once. With an issue rate of four and a single cache port, performance is improved by 3.7%. With two data cache ports this figure rises slightly to 3.95%. Therefore, although loop unrolling improves the performance of two benchmarks, quick and puzzle, by almost 10%, it has a relatively small overall impact on HARP performance.

7. CONCLUSIONS

This paper has described RLS, a resource limited scheduler, which has been used to evaluate iHARP, a VLIW processor with an instruction issue rate of four. With four

pipelines a speedup of 1.80 was achieved using non-numeric benchmarks. Significantly, this performance was based on an architecture which has been implemented in silicon, and not on an abstract model. Also the speedup recorded is relative to optimised single pipeline code where branch delay slots have been filled wherever possible.

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

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

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

Comprehensive comparisons with alternative addressing mechanisms confirm that our decision to incorporate ORed indexing in iHARP was fully justified and provided a 10% boost in performance. Surprisingly, however, register indirect addressing performed equally well, although requiring 10% more instructions. Given the complex stack alignment requirements of ORed indexing, register indirect addressing is therefore a strong candidate for future architectures, in spite of the additional code overhead.

Other results suggest that restricting the register file to two write back ports did not significantly degrade iHARP performance, that RLS was not able to make significant use of the two parallel branch units in iHARP and that instruction combining was not a critical factor in iHARP processor performance. Finally, the limited loop unrolling performed by RLS resulted in only modest performance improvements.

ACKNOWLEDGEMENTS

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

REFERENCES

1. Johnson M "Superscalar Microprocessor Design," Prentice Hall, 1991.
2. Wang L "Instruction Scheduling for a Family of Multiple-Instruction-Issue Architectures," PhD Thesis, University of Hertfordshire, December 1993.
3. Nicolau A "Uniform Parallelism Exploitation in Ordinary Programs," Proceedings of International Conference on Parallel Processing, August 1985, pp 614-618.
4. Wall D W "Limits of Instruction-Level Parallelism," ASPLOS IV, April 1991, pp 176-188.
5. Butler M, Yeh Tse-Yu, Patt Y, Alsup M, Scales H and Shebanow M "Single Instruction Stream Parallelism is Greater than Two," 18th Annual International Symposium on Computer Architecture, Toronto, May 1991, pp 276-286.
6. Lam M S and Wilson R P "Limits of Control Flow on Parallelism," 19th Annual International Symposium on Computer Architecture, Gold Coast, Australia, May 1992, pp 46-57.
7. 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.
8. Fisher J A "Trace Scheduling: A Technique for Global Microcode Compaction," IEEE Transactions on Computers, C-30, (7), July 1981, pp 478-490.
9. 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," 18th Annual International Symposium on Computer Architecture, Toronto, May 1991, pp 266-275.
10. Mahlke S A, Lin D C, Chen W Y, Hank R E and Bringmann R A "Effective Compiler Support for Predicated Execution using the Hyperblock," Micro25, Portland, Oregon, December 1992, pp 45-54.
11. Lam M S "Software Pipelining: An Effective Scheduling Technique for VLIW Machines," SIGPLAN88 Conference of Programming Language Design and Implementation, Georgia, USA, June 1988, pp 318-328.
12. Moon S and Ebcioglu K "An Efficient Resource-Constrained Global Scheduling Technique for Superscalar and VLIW Processors," Micro25, Portland, Oregon, December 1992, pp 55-71.
13. 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.
14. Stallman R M "Using and Porting GNU CC," Free Software Foundation, 1989.
15. Whale D J "Development of a Processor Simulation for iHARP," Division of Computer Science, University of Hertfordshire, April 1992.
16. Smith M D, Horowitz M and Lam M "Efficient Superscalar Performance through Boosting," ASPLOS V, October 1992, pp 248-259.
17. Hsu P Y T and Davidson E S "Highly Concurrent Scalar Processing," 13th Annual International Symposium on Computer Architecture, 1986, pp 386-395.
18. Furber S "VLSI RISC Architecture and Organization," Marcel Dekker, 1989.

19. Hennessy J L and Patterson D A "Computer Architecture: A Quantitative Approach," Morgan Kaufmann, San Mateo, California, 1990.
20. Steven G B "A Novel Effective Address Calculation Mechanism for RISC Microprocessors," SIGARCH, 16, (4), 1988, pp 150-156.
21. Glew A "ORed Indexing," personal communication, April 1989.
22. Abnous A and Bagherzadeh N "Architectural Design and Analysis of a VLIW Processor," U.C., Irvine, Technical Report No. 92-79, October 1992.

Fig 1 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 2 can use a 32-bit literal from an adjacent pipeline.

Fig 2 The RLS Algorithm

```
proc: RLS(program, options)
    for each procedure in the program do
        Read in the sequential code;
        Construct a flow graph;
        Detect loops and unroll simple loops;
        while unscheduled nodes do
            Best-node := select(flow graph);
            Perform bookkeeping;
            Determine the schedule class for best-node;
            /*low-level scheduling begins*/
            for each instruction i in best-node do
                Attempt to percolate instruction i into 'window' of already
                scheduled instructions;
                if percolation fails then
                    Append a new long instruction word to the 'window';
                    Insert i into the new long instruction word;
                end if
            end for;
        end while;
    end for;
    output scheduled code;
end proc
```

Fig 3 Speedup versus Instruction Issue Rate

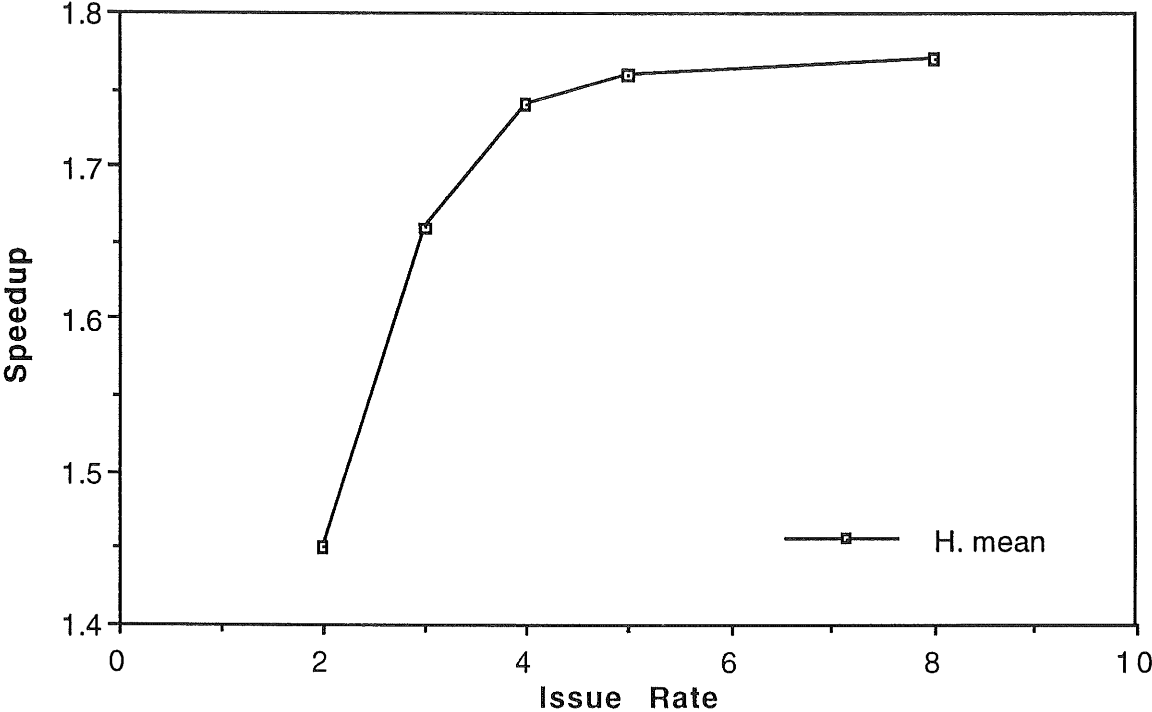


Fig 4 Code Size versus Instruction Issue Rate

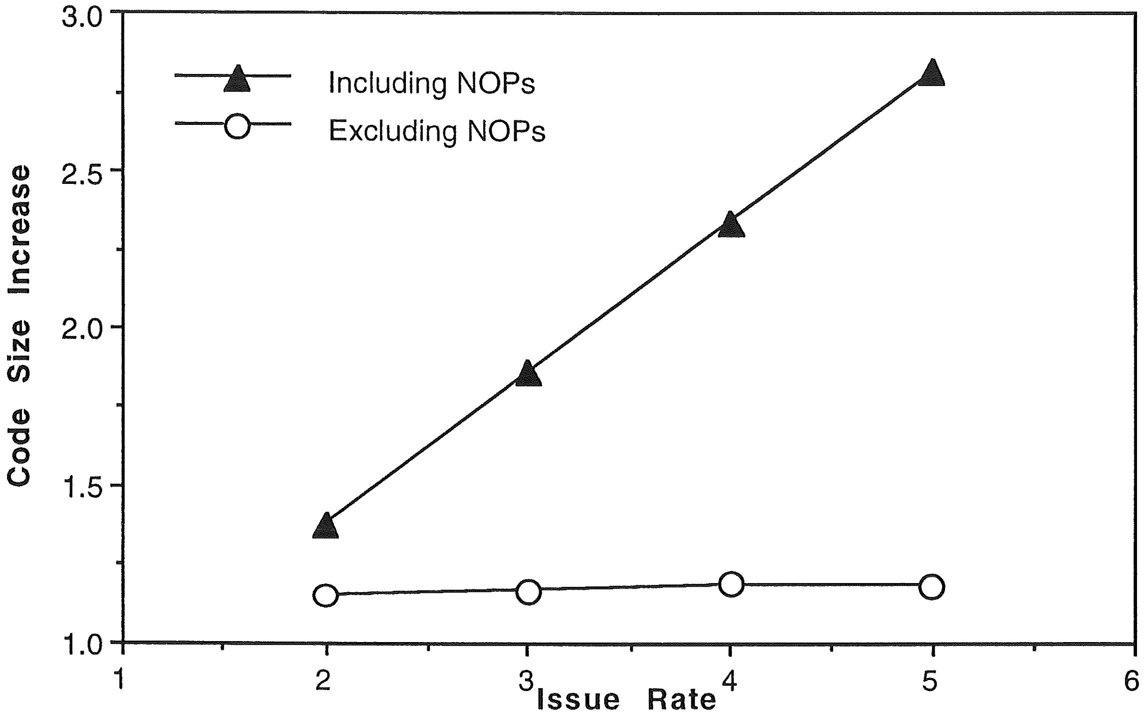


Fig 5 Speedup versus Number of Data Cache Ports

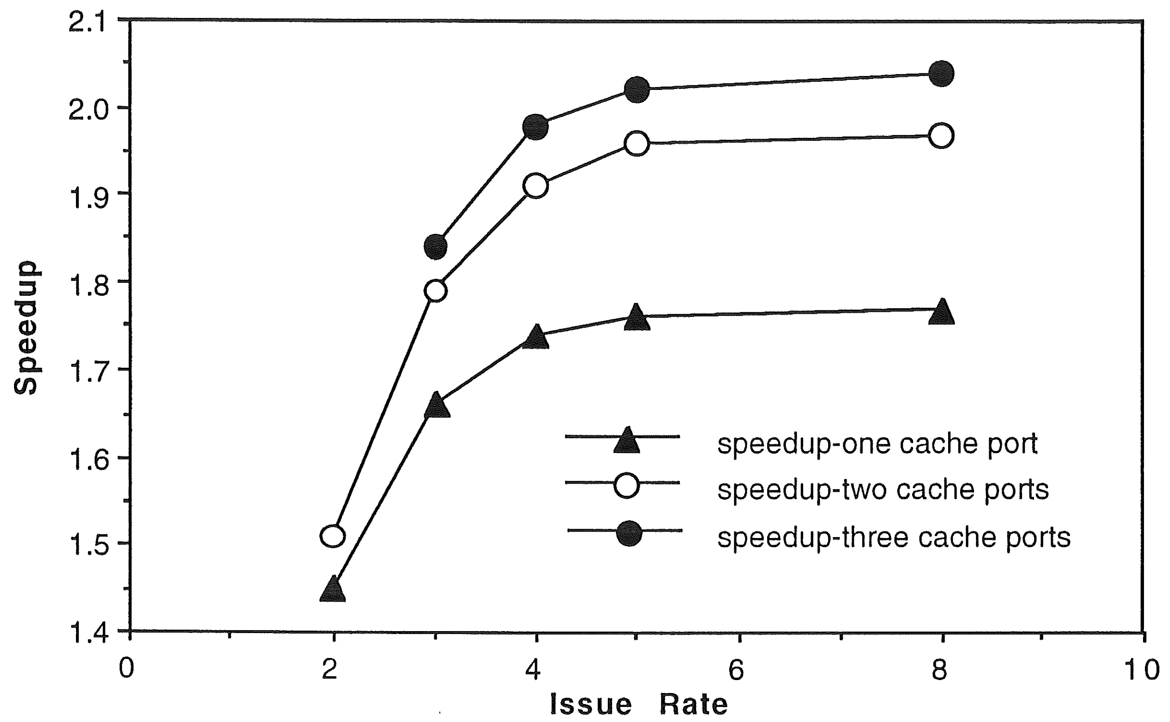


Fig 6 Speedup versus Number of Cache Ports using Inter-procedural Scheduling

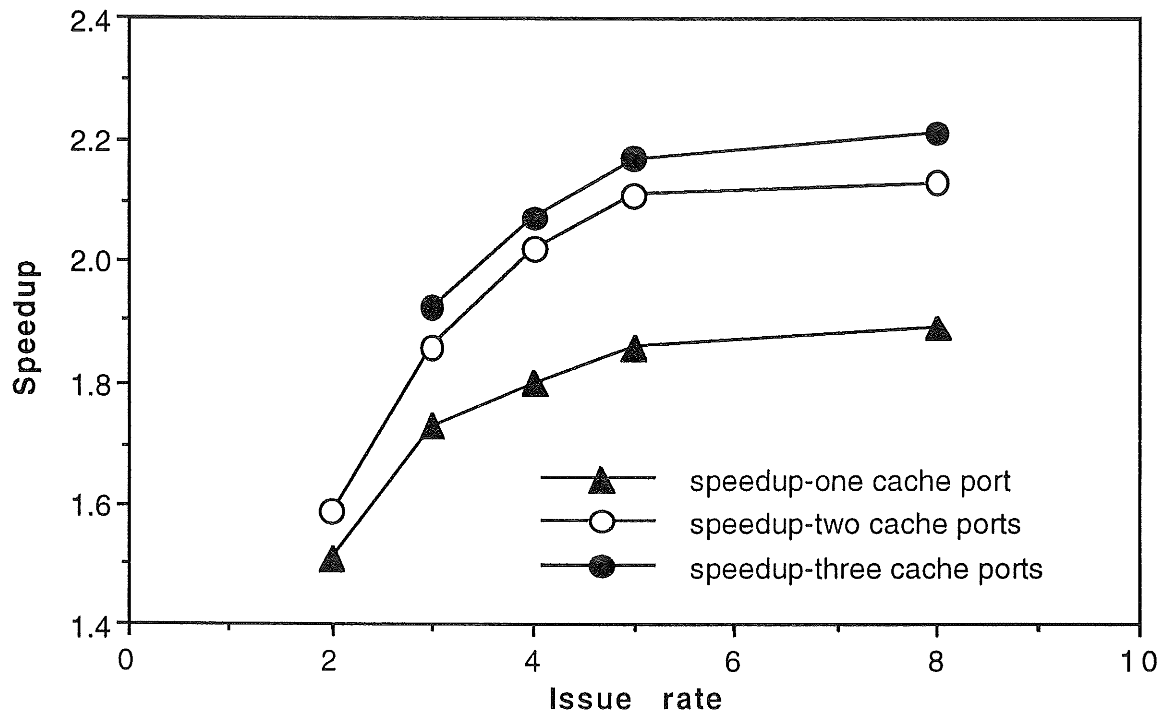


Fig 7 Impact of Guarded Execution using One Memory Port

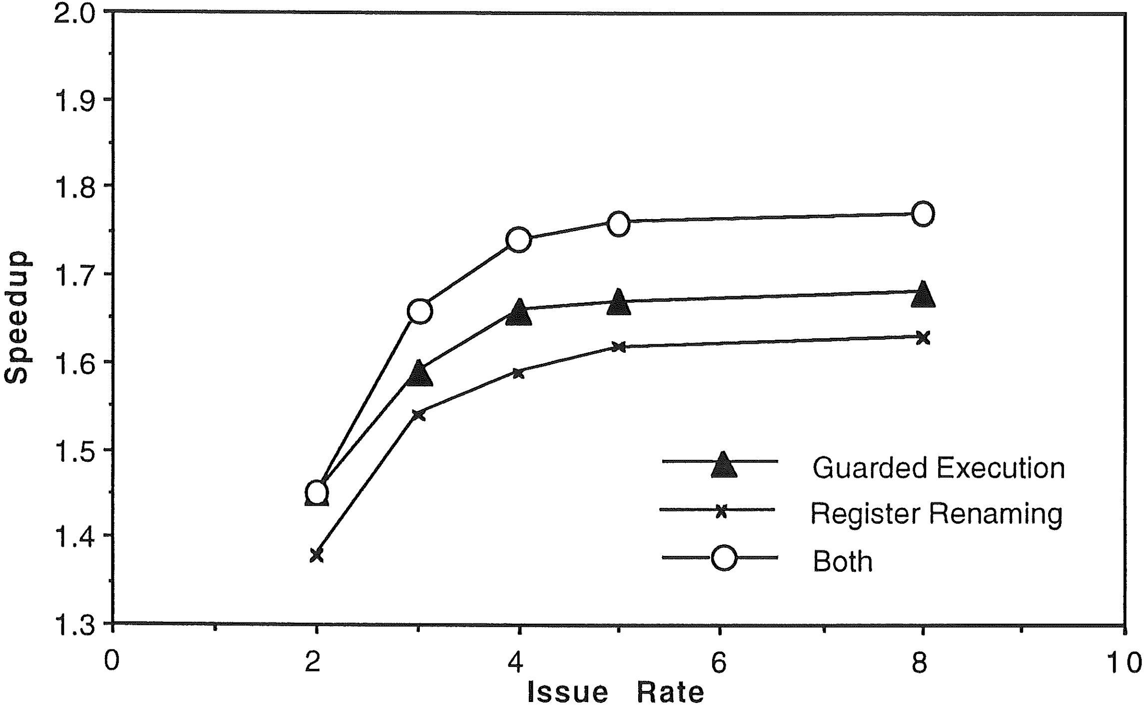


Fig 8 Impact of Guarded Execution using Two Memory Ports

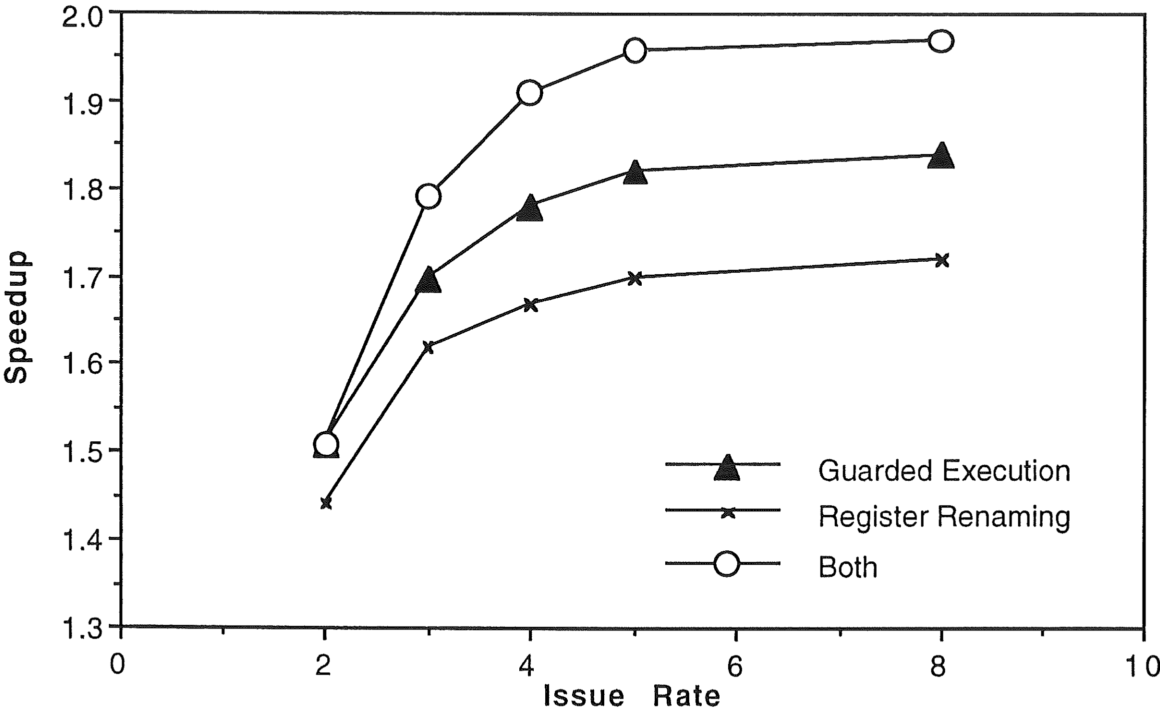


Fig 9 Relative Performance of Addressing Mechanisms