DIVISION OF COMPUTER SCIENCE

# Using Conditional Execution to Exploit Instruction Level Concurrency

Sue Gray
Rod Adams

Technical Report No.181

March 1994

**Using Conditional Execution to Exploit Instruction Level Concurrency**

**SUE GRAY AND ROD ADAMS**
**Department of Computer Science, University of Hertfordshire, Hatfield, AL10 9AB, UK.**

## SUMMARY

Multiple-instruction-issue processors seek to improve performance over scalar RISC processors by providing multiple pipelined functional units in order to fetch, decode and execute several instructions per cycle. The process of identifying instructions which can be executed in parallel and distributing them between the available functional units is referred to as instruction scheduling. This paper describes a simple compile-time scheduling technique, called conditional compaction, which uses the concept of conditional execution to move instructions across basic block boundaries. It then presents the results of an investigation into the performance of the scheduling technique using C benchmarks programs scheduled for machines with different functional unit configurations.



KEY WORDS Static Instruction Scheduling Conditional Execution Conditional Compaction Resource Configurations


## INTRODUCTION

Multiple-instruction-issue machines seek to increase processor performance by exploiting the low level parallelism available in compiled code. Execution rates in excess of one instruction per cycle are achieved by providing multiple pipelined functional units in order to fetch, decode and execute several instructions per cycle. Instruction level concurrency is detected dynamically by the hardware, statically by the compiler, or by a combination of the two techniques. In general multiple-instruction-issue processors are categorised as superscalar or very long instruction word (VLIW) machines.[1]

Superscalar processors are scheduled dynamically, with or without assistance from the compiler. A typical superscalar processor fetches instructions from the instruction cache into a buffer, referred to as a window. Complex instruction issue logic is then used to select instructions from the window for parallel issue to the appropriate functional units. In contrast VLIW machines rely solely on the compiler to detect parallelism. The functional units in a VLIW processor are controlled by concurrent instructions which are packed into long instruction words (LIWs) by the compiler. Long instruction words are then issued, in order, at a rate of one LIW per cycle. In this paper we are concerned with a simple VLIW scheduling technique, called conditional compaction, which produces long instruction words for a machine model with variable resource configurations.

The ability of the scheduler to exploit the parallelism available within an application is fundamental to the performance of a VLIW machine. The amount of parallelism available within an application and the amount of parallelism which is realised by a particular scheduling technique is largely dependent on the nature of the

application.

Numeric applications are characterised by a high ratio of computations to dynamic branches. Furthermore, a large percentage of the branches are do-loop branches which can be resolved early. Such programs respond well to techniques such as trace scheduling[2] which relies heavily on program profiling to predict the most likely path through a program; and to techniques such as loop unrolling[3] and software pipelining[4] which concentrate on exposing the parallelism between successive loop iterations.

Non-numeric applications are characterised by a high proportion of data dependent branches, small loop bodies, and low loop iteration counts. Hence such programs respond well to techniques such as boosting[5], enhanced pipeline scheduling[6] and conditional compaction which focus on removing the dependencies caused by data dependent branch instructions.

Boosting is based on the concept of speculative execution. Speculative execution is supported by "shadow" structures which are used to buffer the results of instructions which are "boosted" across conditional branches. These results are then committed or squashed when the outcomes of the preceding branches are known. Enhanced pipeline scheduling and conditional compaction are based on the concept of conditional execution. An instruction which is moved across a conditional branch from either the branch target or the sequential execution path is conditionally executed on the value of the branch condition which results in execution along that path.

In the following sections we describe the architectural and software framework for the development of the conditional compaction technique, give a detailed description of the algorithms used by the scheduler, and present the results of an investigation into the performance of the technique for target architectures with different functional unit configurations.

## ARCHITECTURAL AND SOFTWARE FRAMEWORK

The conditional compaction algorithm was developed in conjunction with the HARP architecture.[7,8] HARP is a statically scheduled multiple-instruction-issue architecture which is characterised by multiple functional units and a conditional execution mechanism. Conditional execution is supported by a set of one-bit Boolean registers. There are four types of instruction: ALU (i.e. computational and relational), memory reference, Boolean, and branch. All instructions may be conditionally executed on the value of a Boolean register specified in the instruction. For example the instruction

F B3 ADD R20, R19, #4

is executed if and only if the Boolean register B3 contains the value FALSE.

The architecture has a compact four stage pipeline (see Figure 1) which combined with unrestricted register bypassing results in an operational latency of one cycle for all instructions except MULT, DIV and load a return address. Delays are implemented using dummy WAIT instructions which use the same resources as the parent

instructions generating the delay. The Boolean registers are set by relational and Boolean instructions and tested by conditional branch instructions. This two instruction branch architecture results in a branch delay of one cycle.

Two addressing modes are provided, register indirect with index, and register indirect with displacement. The architecture uses an ORed addressing mechanism which removes the need for an addition in an address calculation and allows the ALU and memory reference stages of the pipeline to be combined. Memory addresses are computed by ORing the two address components[9] in the RF stage of the pipeline. The compiler guarantees that an OR operation is equivalent to an addition by ensuring that the base register always contains a multiple of a power of two and that the offset/index is always less than this power of two.

Multiple ALUs and address units, a Boolean unit and a PC unit allow the execution of several computational, relational, and memory reference instructions in parallel with a single Boolean instruction and a maximum of two branch instructions. The model provides 64, 32-bit, general purpose registers and 32, 1-bit, Boolean registers.

Our experiments were conducted using a simulation of the machine model to run a set of conditionally compacted C benchmarks. The simulator allows the user to specify the total number of instructions contained in a long instruction word. The benchmark programs were compiled into optimised sequential code using a C compiler generated by the GNU-CC compiler generator. The compiler generator takes a machine description as input and produces a C compiler for the specified architecture. The resulting compiler provides a comprehensive range of classical optimisations and an optional procedure in-lining facility.

The sequential code for each C function is packed into long instruction words by the instruction scheduler. Each long instruction word consists of a fixed number of branch, ALU, boolean and memory reference slots. Slots which are not filled by the scheduler are packed with NOPs.The compaction process is divided into two phases. First a local compaction algorithm is used to schedule the instructions within each basic block (i.e. within sections of code which can only be entered at the beginning and exited at the end); then the conditional compaction algorithm is used to extend the scope of the scheduler across the whole function by moving conditionally executed instructions into the empty slots in the locally compacted code.

## LOCAL COMPACTION

The process of scheduling unconditionally executed instructions within a basic block is called local compaction. Local compaction consists of two phases: constructing a DAG, called a data interaction graph (DIG), to represent the partial ordering of instructions which is maintained by the scheduler in order to preserve data integrity, and forming long instruction words from sets of potentially concurrent instructions identified using

the graph.

The local compaction program builds a list of long instruction words (LIWs) one long instruction at a time, in sequential order. The LIW which is currently under construction is called the current long instruction word (CLIW). The set of instructions which can be scheduled in the CLIW without violating data integrity is referred to as the data available set. List scheduling is used to schedule an instruction from the data available set into the CLIW, subject to resource limitations. The data available set is then recomputed and the process is repeated until the CLIW is complete.

The definition of data availability is given in terms of the following relationships between instructions.

*Definition* 1: Strong Data Interaction

Given two instructions $s_i$ and $s_j$, where $s_i$ precedes $s_j$ in the sequential code, $s_i$ and $s_j$ are said to have a strong data interaction if they satisfy any of the following conditions.

1. $s_i$ defines a register or memory location used by $s_j$. This is termed a definition versus use constraint.
2. $s_i$ and $s_j$ define the same register or memory location. This is termed a definition versus definition constraint.
3. $s_i$ uses a memory location defined by $s_j$. This is termed a use versus definition constraint with respect to data held in memory.

*Definition* 2: Weak Data Interaction

Given two short instructions $s_i$ and $s_j$, where $s_i$ precedes $s_j$ in the sequential code, $s_i$ and $s_j$ are said to have a weak data interaction if they satisfy both the following conditions.

1. There is no strong data interaction between $s_i$ and $s_j$
2. $s_i$ uses a register defined by $s_j$. This is termed a use versus definition constraint with respect to data held in a register.

*Definition* 3: Data Available

An instruction $s_j$ is said to be data available with respect to the CLIW if $s_j$ satisfies both the following conditions.

1. Every short instruction in the basic block which precedes $s_j$ in the sequential code and has a strong data interaction with $s_j$ appears in a long instruction which precedes the CLIW in the list.
2. Every short instruction in the basic block which precedes $s_j$ in the sequential code and has a weak interaction with $s_j$ appears in a long instruction which

precedes the CLIW in the list, or appears in the CLIW itself.

Parallel execution is possible between instructions which have a weak data interaction (i.e. a register use versus definition constraint) as data is read from the register file in the register fetch stage of the pipeline, one cycle before the new value is computed in the ALU stage.

The data available set is computed using the information in the data interaction graph. The DIG for a block is a DAG, where nodes represent instructions, and node A is a parent of node B means that node A precedes node B in the sequential code and node A has a data interaction with node B. Arcs are labelled to indicate strong and weak data interactions. The DIG for a block is implemented by a labelled adjacency matrix which is constructed by scanning backwards through the block comparing each instruction to each of its predecessors.

Detecting an interaction between two instructions with respect to the data held in a register is easily achieved by comparing the input and output registers of both instructions. Detecting a data interaction with respect to data held in memory is a more complex problem known as memory reference disambiguation.[2] Since memory addresses are calculated from two components, the local compaction program must assume that there is a data interaction with respect to two items of data held in memory, unless it can be shown that the sum of their respective address components are not equal. For example there is no interaction between the instructions

LD R21, (R22, R23)          ST 6(R25), R24

with respect to the data held in registers, but if (R22, R23) and 6(R25) could possibly specify the same address there is use versus definition constraint between the instructions with respect to the data held in memory.

Storage for each invocation of a C function is allocated dynamically on the run-time stack. The C compiler computes the size of each stack frame and uses the stack pointer to maintain activation records and access data. Local variables and actual parameters are referenced using positive offsets from the stack pointer which points to the top of the stack, and the current activation record is popped from the stack by adjusting the stack pointer by the stack frame size. Thus the contents of the stack pointer remains constant throughout the invocation of a function, and two addresses which are specified using different offsets from the stack pointer are guaranteed to be distinct.

Hence given two memory reference instructions $s_i$ and $s_j$, where $s_i$ precedes $s_j$ in the sequential code, $s_i$ and $s_j$ are said to have no data interaction with respect to data held in memory if $s_i$ and $s_j$ are of the form

| | | | |
|---|---|---|---|
| | ST $offset_i(Rsrc1_i)$, $Rsrc2_i$ | and | LD $Rdst_j$, $offset_j(Rsrc1_j)$ |
| or | ST $offset_i(Rsrc1_i)$, $Rsrc2_i$ | and | ST $offset_j(Rsrc1_j)$, $Rsrc2_j$ |
| or | LD $Rdst_i$, $offset_i(Rsrc1_i)$ | and | ST $offset_j(Rsrc1_j)$, $Rsrc2_j$ |

and $Rsrc1_i = Rsrc1_j = SP$ and $offset_i \neq offset_j$

Otherwise two memory reference instructions $s_i$ and $s_j$ are said to have a strong data interaction, with respect to data held in memory, if they satisfy any of the following conditions.

1. $s_i$ is a ST instruction and $s_j$ is a LD instruction
   (implies a definition versus use constraint with respect to data held in memory)
2. $s_i$ is a ST instruction and $s_j$ is a ST instruction
   (implies a definition versus definition constraint with respect to data held in memory)
3. $s_i$ is a LD instruction and $s_j$ is a ST instruction
   (implies a use versus definition constraint with respect to data held in memory)

During the construction of a list of long instruction words from the instructions in a basic block, the local compaction program uses the DIG for the block to compute the set of instructions which are data available with respect to the current long instruction word. List scheduling[10] is then used to determine the order in which instructions from the data available set are considered for inclusion in the CLIW. List scheduling is a heuristic technique wherein each instruction is assigned a priority prior to scheduling. Then, given a set of available instructions, only the "best" long instruction word is formed by scheduling the instructions with the highest priorities. In our scheduler each instruction is assigned a scheduling priority which reflects its position in the sequential code. The data available set is implemented as a list of instructions which are ordered in decreasing magnitude of scheduling priority. The local compaction program then attempts to schedule the short instruction from the head of the list into the CLIW, subject to the resource limitations of the target architecture.

*Definition* 4: Resource Available
An instruction $s_j$ which requires a functional unit U is said to be resource available with respect to the CLIW if the number of instructions already scheduled in the CLIW which require a functional unit U is less than the total number of this type of functional unit provided by the target machine.

An instruction $s_j$ which is data available with respect to the CLIW can only be scheduled in the CLIW if it is resource available with respect to the CLIW.

The local compaction program schedules the instructions in a basic block into long instruction words using the algorithm given in Figure 2. Since the HARP pipeline allows two instructions which have a register use-definition dependency to be

scheduled in parallel, the local compaction program updates the data available set each time it adds a new instruction to the CLIW.

Branch instructions determine a program's flow of control and, by definition, a branch instruction must occur at the end of a basic block. HARP has a branch delay of one cycle, hence a branch instruction must be scheduled in the penultimate long instruction word of a compacted block. The DIG represents the ordering constraints on a branch instruction resulting from data dependencies, but cannot be used to represent the control dependencies which relate to long instruction words. Hence a branch instruction and its associated NOP are scheduled last, and are packed into the existing penultimate, last or newly created LIWs, depending on data and resource availability.

## CONDITIONAL COMPACTION

Conditional compaction is a simple technique which uses the concept of conditional execution to extend the scope of the scheduler across a whole function or procedure. The conditional compaction program attempts the fill the empty slots in a block of locally compacted code with instructions from the branch target and sequential execution paths. Instructions which are moved across a conditional branch are conditionally executed on the value of the branch condition (i.e. the Boolean variable) which would result in their execution. This allows instructions to be moved from both instruction streams without the need for global data flow analysis or a branch prediction scheme.

The conditional compaction program builds a flow graph for the basic blocks in a function, and classifies each block according to the nature of the branch instruction it contains (see Figure 3). The blocks which are candidates for conditional compaction are held in a "compaction" list. The conditional compaction program repeats the process of removing and conditionally compacting the block at the head of the list until the list is empty. Initially only type 2 and type 3 blocks (i.e. blocks which end with an unconditional or conditional branch) are candidates for conditional compaction. These blocks are placed in the compaction list in sequential code order. Thereafter any block not present in the list is added to the head of the list whenever the compaction process results in the movement of instructions which may permit further compaction to take place.

The block at the head of the list is removed for conditional compaction and is referred to as the C_block. If the C_block is type 2 the compaction program attempts to move instructions from the branch target block into the C_block's locally compacted schedule. If the C_block is type 3, and the branch is forwards, the scheduler favours the removal of short branches by attempting to move instructions from the sequential successor block, before considering the branch target block. If the branch is backwards the scheduler favours the compaction of inner loops by considering the branch target block before the sequential successor block.

Given a C_block and its successor the scheduler computes the set of instructions from the successor block, excluding a branch, which could possibly be moved into the C_block. This is referred to as the conditionally available set (CASet). For a type 2 block the CAset contains a copy of the branch target's sequential code up to, but not including, a branch. For a type 3 block the CASet consists of conditionally executed copies of those instructions from the successor block, excluding a branch, which can be conditionally executed on the value of the Boolean variable which determines the C_block's branch. The scheduler merges the CAset into the sequential code for the C_block, to form a single unit, and uses a variation on the local compaction algorithm given in Figure 2 to schedule instructions from the CASet into the C_block's locally compacted schedule. The instructions corresponding to those which are successfully scheduled in the C_block are removed from the successor, and the remaining sequential code is locally rescheduled.

If instructions are moved out of a successor block which can be reached from more than one predecessor (i.e. from blocks other than the C_block) then compensation code must be introduced on to the alternative execution paths to ensure the correctness of the code. Hence the scheduler only attempts to move instructions from a sequential successor block if it cannot be reached from any other block (i.e. the sequential successor is not a branch target). This avoids the introduction of a branch over the compensation code, and is the reason type 1 blocks are never considered for conditional compaction.

**Scheduling Branch Instructions in Parallel**

If the conditional compaction process succeeds in moving all the non-branch instructions from a branch target or sequential successor block which is type 2, 3 or 4 it may be possible to move the remaining branch instruction into the C_block's penultimate LIW. It may also be possible to move a BSR instruction from an otherwise empty sequential successor block, but a BSR instruction cannot be moved from a branch target block, as this would result in the processor saving the incorrect return address.

Moving branches across basic block boundaries results in several new types of block, which can also be candidates for conditional compaction. The nature of the new block depends on the type of the C_block, the type of the successor, and whether the successor is a sequential successor or a branch target. Figure 4 shows the nature of the blocks which result when branches are moved out of sequential successor blocks. Figures 5 and 6 show the nature of the blocks which result when branches are moved from the targets of unconditional and conditional branches.

If the conditional compaction process results in a block obtaining a new sequential successor, or a new branch target, the block is returned to the compaction list for repeated scheduling. For example, Figure 7 shows the locally compacted HARP code

for the statement

```
if ( Trial(k) || (k == 0) )
    return(true)
else
    Remove(i,j)
```

which is taken from the body of a for loop found in the Trial function of the puzzle benchmark. Blocks 1 to 5 represent the if then else statement, and the last block contains the housekeeping code to exit the function. Blocks 2, 3 and 4 are held in the compaction list. Figure 8 shows the result of conditionally compacting block 2. Referring to the locally compacted code (Figure 7): block 2 branches forwards so its sequential successor block, block 3, is considered before its branch target block. The scheduler computes the CASet which is {F B6 NE B7, R17, #0}, schedules this instruction in parallel with block 2's branch (BT B6, Lab33), and removes the corresponding NE instruction from block 3. The remaining branch (BT B7, Lab32) cannot be moved into block 2's penultimate long instruction word; and block 3 can only be entered from block 2, so no compensation code is required. The scheduler then considers the branch target block, block 4. Again the scheduler computes the CASet, which is {T B6 MOV R5, #1}, schedules this instruction in parallel with block 2's branch (BT B6, Lab33), and removes the corresponding MOV instruction from block 4. However, in this case it is possible to move the remaining branch (BRA Lab26) into block 2's penultimate long instruction word (see Figure 6 (i)), resulting in the single branch instruction (BT B6, Lab26) in block 2. All the instructions in block 4 have now been moved into block 2, but block 4 is also block 3's sequential successor, so a copy of block 4 is required as compensation code.

Block 2 now has a new branch destination block so it is returned to the compaction list where it is reselected as the C_block, and the process is repeated. The first five instructions from the new branch destination (the last block) are successfully scheduled in block 2 on the condition T B6. These instructions are removed from the end block, and block 2's branch destination is adjusted accordingly (Lab411). The end block has several other predecessors besides block 2, so a block of compensation code, containing the scheduled instructions, is introduced as the end block's sequential predecessor. Block 2's compaction is now complete, resulting in the code given in Figure 8.

Finally Figure 9 shows the result of conditionally compacting block 3. Referring to the partially compacted code given in Figure 8: block 3 branches forwards so the new block of compensation code is considered before the branch target block. The instruction MOV R5, #1 is scheduled in parallel with block 3's branch (BT B7, Lab32) on the condition F B7. The new block's branch, BRA Lab26, is then moved into block 3's penultimate long instruction word (see Figure 4(i)), on the condition F B7,

resulting in a block with two branches BT B7, Lab32 and BF B7 Lab26 (the unconditionally executed equivalent of F B7 BRA Lab26). The block of compensation code could only be entered from block 3, so block 5 is now block 3's sequential successor. Block 3's first branch instruction BT B7, Lab32 is thus redundant, and is removed, leaving block 3 with a new sequential successor, block 5, and a new branch target block (the last block's compensation code). Block 3 is then returned to the compaction list and the process is repeated. All the instructions in block 5 are scheduled in block 3, including the branch (see Figure 4(iii)). Block 5 can only be entered from block 3, so no compensation code is introduced. All the instructions in the last block's compensation code are scheduled in block 3 on the condition F B7, and the branch destination is adjusted accordingly (Lab411), but the compensation code is still required as it still has other predecessors. Block 3's compaction is now complete, resulting in the code given in Figure 9.

The sequential code for the if then else statement consists of sixteen instructions. This is reduced to twelve long instruction words in the locally compacted code, and seven long instructions words in the final conditionally compacted code. Hence there is a saving of nine cycles over the sequential code, and five cycles over the locally compacted code, each time the outer for loop containing the conditionally compacted if then else statement is executed.

## INVESTIGATION

The investigation was to study the speedup of conditionally compacted code, scheduled for multiple-instruction-issue models with different function unit configurations, over the equivalent sequential code running on a single-instruction-issue model.

### The Benchmarks

The experiments were conducted using the C versions of the Stanford integer benchmarks[11] running on a simulation of the HARP model which allows the user to specify the total number of instructions contained in a long instruction word (subject to a maximum of two branches). The benchmarks were compiled into sequential HARP code using the GNU-CC generated compiler, and the conditional compaction algorithm was used to fill 52.5% of the branch delay slots with instructions from the branch destinations and sequential successor blocks. Table 1 lists the programs and gives the dynamic instruction count for the conditionally compacted sequential code.

### Experimental Parameters and Results

In order to examine the effect of varying the number of branch, memory reference and ALU instructions which can be scheduled in parallel we defined the four base

configurations of the HARP machine model specified in Table 2. We then measured the speedups of the individual benchmarks running on variations of the four base configurations which allow one, two, three or four ALU instructions to be scheduled in parallel with the branch and memory reference instructions. The speedup for a particular benchmark, running on a particular instance of the model, was calculated as the ratio of the dynamic instruction count for the benchmark's conditionally compacted sequential code to the dynamic instruction count for the appropriately scheduled parallel code. Figure 10 shows the harmonic means of the speedups of the eight benchmark programs for the sixteen configurations of the model (the full set of results is given in Appendix 1). These results show that, for this particular set of benchmarks, the maximal performance for all four base configurations is reached when the number of ALU instructions per long instruction word is increased to three.

Comparing the harmonic means of the speedups obtained for the three ALU variations of each of the base configurations. The 1b1m configuration achieves an average speedup of 1.56. Allowing two branch instructions to be scheduled in parallel with one memory reference marginally increases this speedup by a factor of 1.92% to 1.59. However allowing two memory reference instructions to be scheduled in parallel with one branch increases the speedup over the 1b1m model by a factor of 7.69% to 1.68. This result demonstrates the importance of providing at least two data cache ports in the target architecture. Prohibiting the parallel execution of memory reference instructions results in longer locally compacted blocks (since each memory reference instruction requires a long instruction word) and often prevents the scheduler from moving memory reference instructions out of a successor block into the C_block. This in turn prevents the scheduler from taking advantage of the potential for parallel branches provided by the 2b1m model, since the movement of a branch instruction from a successor block is not attempted until the block is otherwise empty.

Finally the 2b2m three ALU configuration of the model achieves a speedup of 1.72 which represents a 10.26% increase in performance over the 1b1m model. This result is encouraging, but the 1.72 speedup obtained for the C benchmarks is significantly less than the speedup of 2.55 obtained for a similar set of Modula-2 benchmarks running on the same configuration of the model.[12] This is partly due to differences in the optimisations performed by the sequential compilers, but is mainly a reflection of the comparative terseness of the C code. In particular C lacks the array bounds checking provided by Modula-2, which leads to shorter basic blocks with less potential for compaction. Given the brevity of the C code it was felt that better results could be obtained by using the procedure inlining facility provided by the GNU-CC generated compiler to remove subroutine calls which inhibit the performance of the scheduler. As with the non-inlined code the maximal performance for all four base configurations of the model was obtained when a maximum of three ALU instructions were scheduled per long instruction word. Table 3 shows the harmonic means of the speedups of the non-inlined and inlined code, obtained using the three ALU variations of the four base

configurations of the model (the full set of results for the three ALU variations of the inlined code are given in Appendix 2). As can be seen from these results procedure inlining increases the speedups obtained for all four configurations, with the 2b2m model achieving a speedup of 1.85 over the equivalent conditionally compacted non-inlined sequential code.

## CONCLUSIONS

This paper describes a global compile-time scheduling technique, called conditional compaction, which is targeted at general-purpose code. The technique, which uses the concept of conditional execution to move instructions across basic block boundaries, is notable for its simplicity. Conditional execution removes the need for global data flow analysis, or a branch prediction scheme, and the algorithm does not incorporate any of the optimisations, such as register renaming or loop unrolling, designed to increase low-level parallelism.

The paper presents the results of an investigation into the speedups obtained for the C versions of the Stanford integer benchmarks scheduled for machines with different functional unit configurations. The study shows that an average speedup of 1.72 is obtained for conditionally compacted non-inlined code scheduled for a "realistic" machine which supports the parallel execution of two branch, two memory reference and three computational instructions. This result is increased to a speedup of 1.85 when procedure inlining is used to increase the potential for parallelism in the sequential code.

## ACKNOWLEDGEMENTS

We wish to thank Dave Whale for his work in implementing the HARP simulation, and Liang Wang and Fleur Steven for producing the sequential C compiler. This work was supported by SERC Research Grant GR/F88018.

## REFERENCES

1. M. Johnson, *Superscalar Microprocessor Design*, Prentice Hall, 1991.
2. J. R. Ellis, *Bulldog: A Compiler for VLIW Architectures*, The MIT Press, Cambridge, Massachusetts, 1986.
3. S. Weiss and J . E. Smith, 'A study of scalar compilation techniques for pipelined supercomputers', *Proc. ACM 2nd International Conference on Architectural Support for Programming Languages and Operating Systems*, 105-109, October 1987.
4. M. Lam, 'Software Pipelining: An effective scheduling technique for VLIW machines', *Proc. ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, 318-327, June 1988.

5. M. D. Smith, M. Horowitz and M. S. Lam, 'Efficient superscalar performance through boosting', *Proc. ACM 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, 248-259, October 1992.
6. S. Moon and K. Ebcioglu, 'An efficient resource-constrained global scheduling technique for superscalar and VLIW processors', *Proc. IEEE 25th Annual International Symposium on Microarchitecture,* 55-71, December 1992.
7. G. B. Steven, S. M. Gray and R. G. Adams 'HARP: A parallel pipelined RISC processor', *Microprocessors and Microsystems*, **13**, (9), 579-587 (1989).
8. R. G. Adams, S. M. Gray and G. B. Steven, 'Utilising low level parallelism in general purpose code: The HARP project', *Microprocessing and Microprogramming*, **29**, (3), 137-149 (1990).
9. G. B. Steven, 'A novel effective address calculation mechanism for RISC microprocessors', *ACM Computer Architecture News*, **16**, (4), 150-156 (1988).
10. S. Davidson, D. Landskov, B. Shriver and P. W. Mallet, 'Some experiments in local microcode compaction for horizontal machines', *IEEE Trans. Comput.*, **C-30**, (7), 460-477 (1981).
11. R. P. Weicker, 'An overview of common benchmarks', *IEEE Computer*, 65-75, December 1990.
12. R. G. Adams, S. M. Gray and G. B. Steven, 'HARP: A statically scheduled multiple-instruction-issue architecture and its compiler', submitted to *2nd Euromicro Workshop on Parallel and Distributed Processing*, Malaga, Spain, January 1994.

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

Figure 1. The HARP instruction pipeline

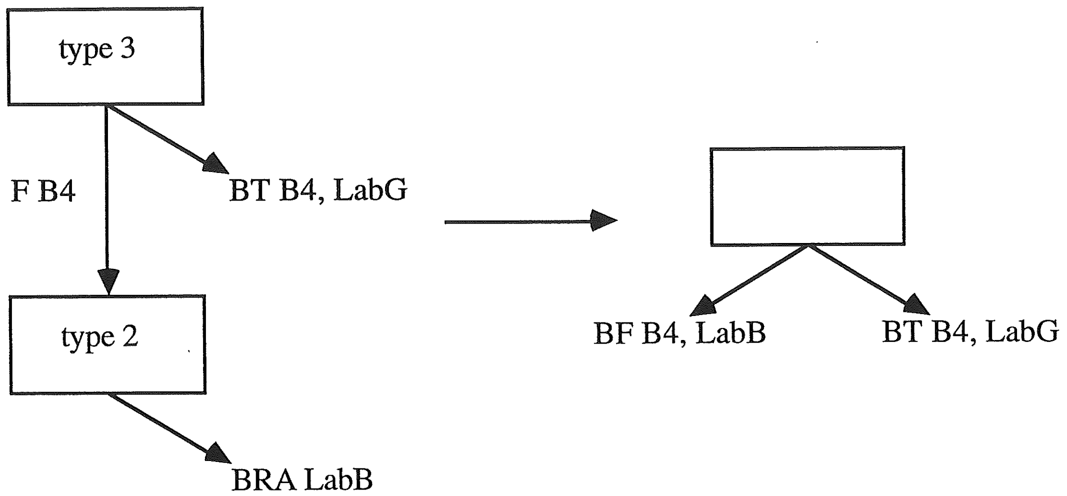
```
WHILE there are still instructions to be scheduled (excluding a branch and NOP) DO
    Generate the current long instruction word (CLIW)
    Compute data available set
    REPEAT
        Find the instruction in the data available set, with the highest priority, which will
        fit into the CLIW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

Figure 2. The local compaction algorithm

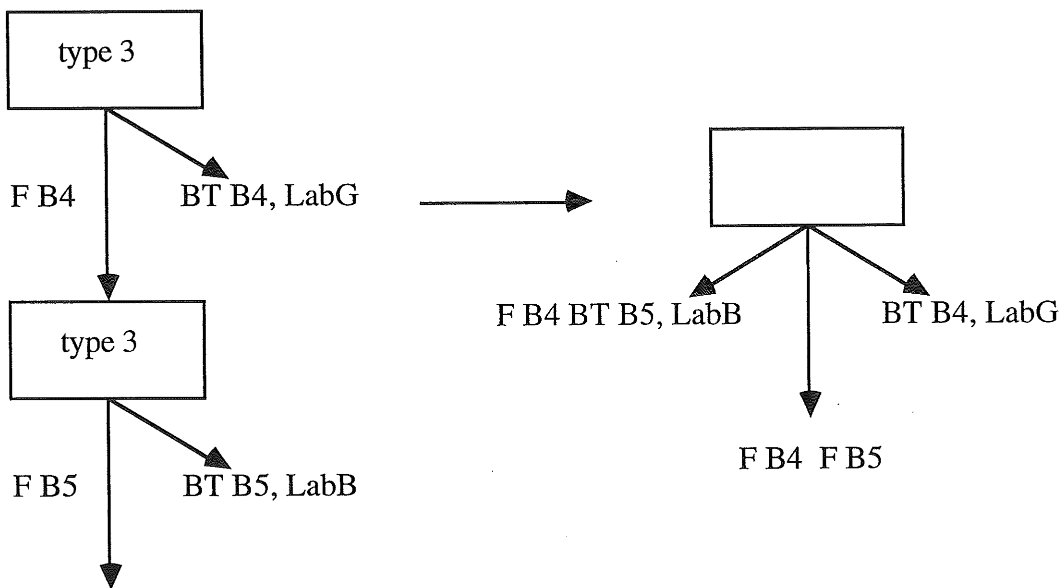
| Block Type | Block Ends With |
|---|---|
| 1 | Any instruction other than a branch |
| 2 | An Unconditional Branch ( BRA ) |
| 3 | A Conditional Branch ( BT or BF ) |
| 4 | Return from Subroutine ( JR ) |
| 5 | Branch to Subroutine ( BSR ) |

Figure 3. Classification of a block according to the nature of its branch

(i) C_block type 3; Sequential Successor Block type 2




(ii) C_block type 3; Sequential Successor Block type 3




(iii) C_block type 3; Sequential Successor Block type 4 or 5

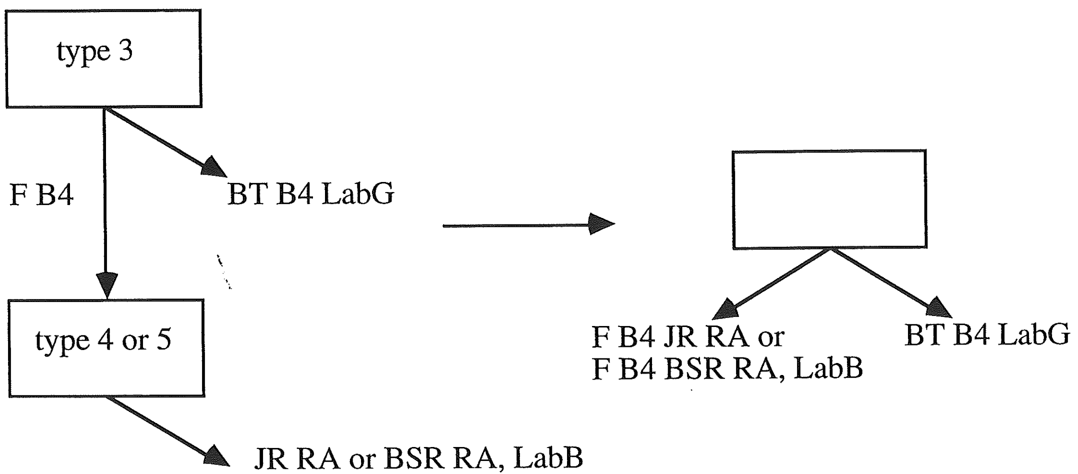
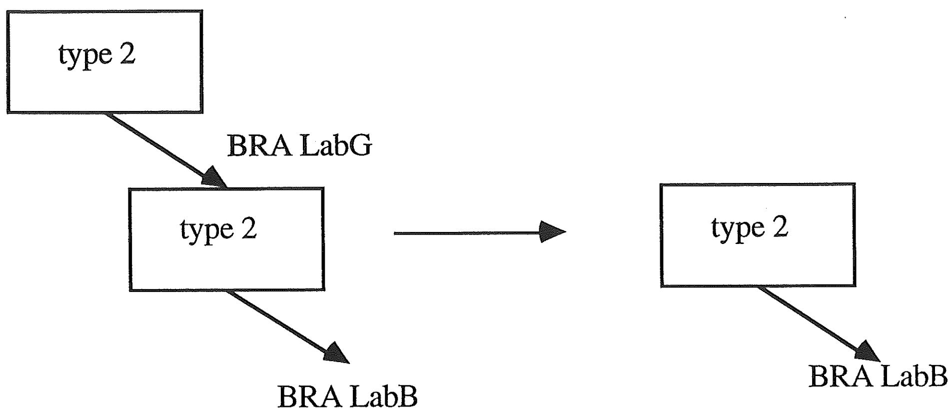
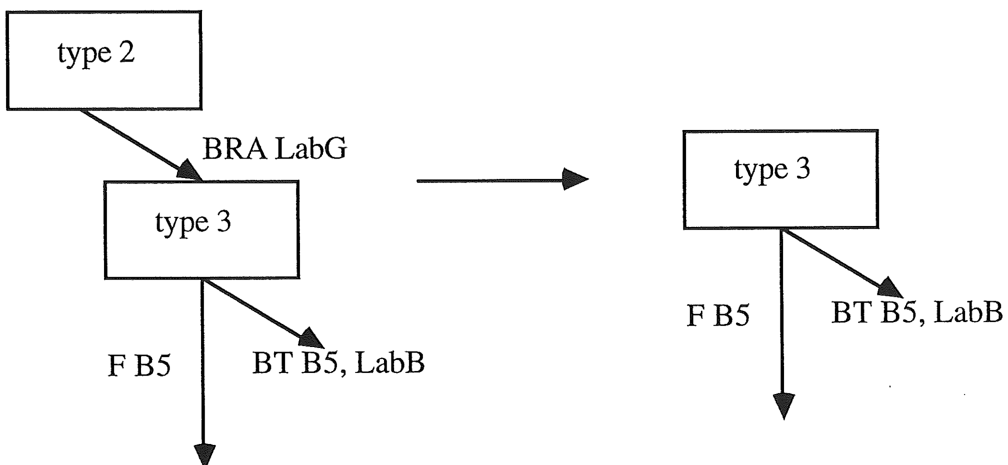



Figure 4. Moving branches from sequential successor blocks

(i) C_block type 2; Branch Target Block type 2




(ii) C_block type 2; Branch Target Block type 3




(iii) C_block type 2; Branch Target Block type 4

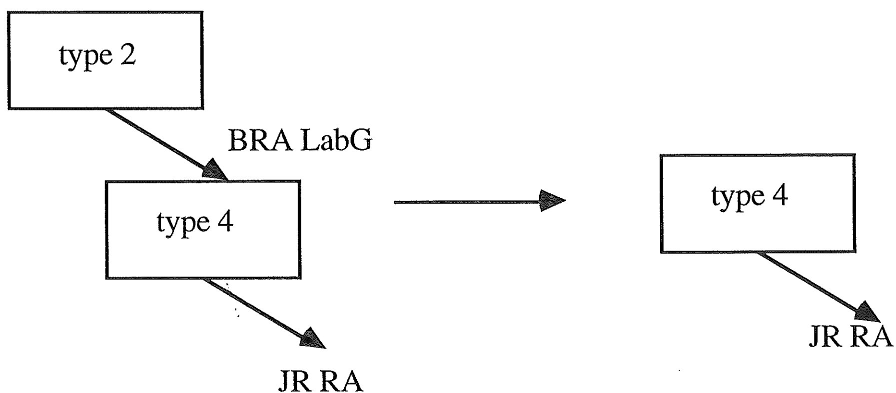
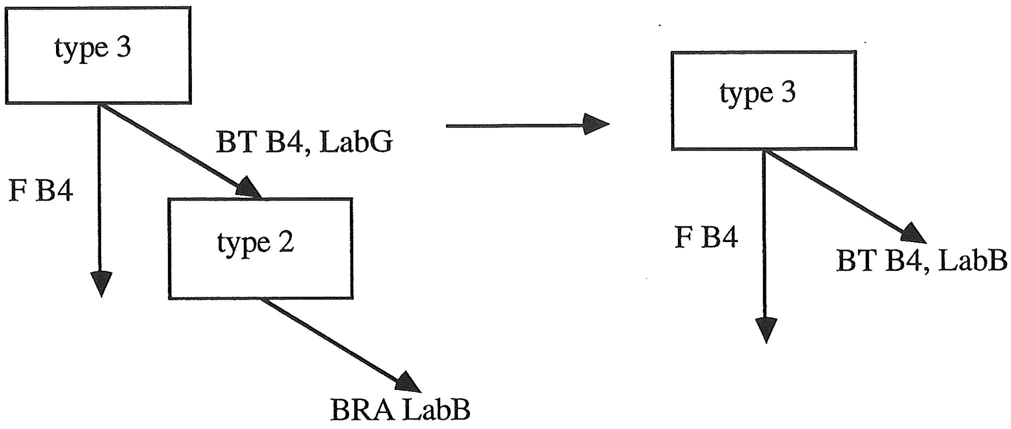
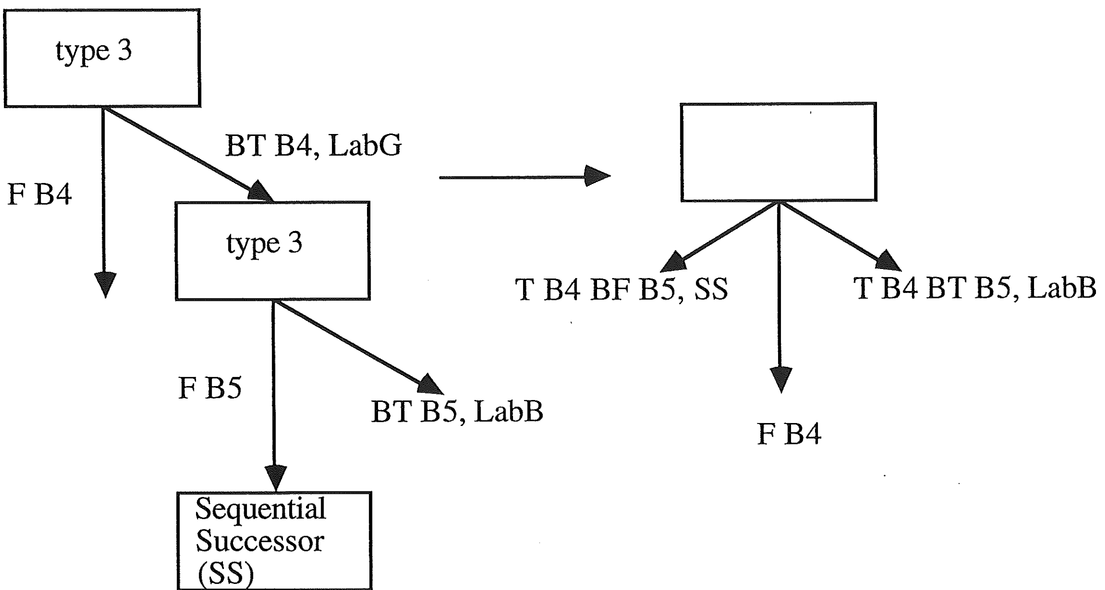



Figure 5. Moving branches from unconditional branch targets

(i) C_block type 3; Branch Target Block type 2




(ii) C_block type 3; Branch Target Block type 3




(iii) C_block type 3; Branch Target Block type 4

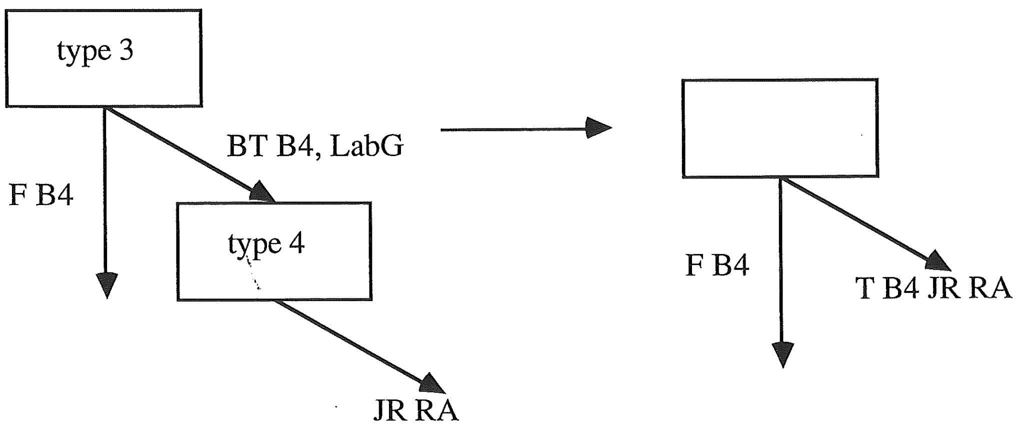



Figure 6. Moving branches from conditional branch targets

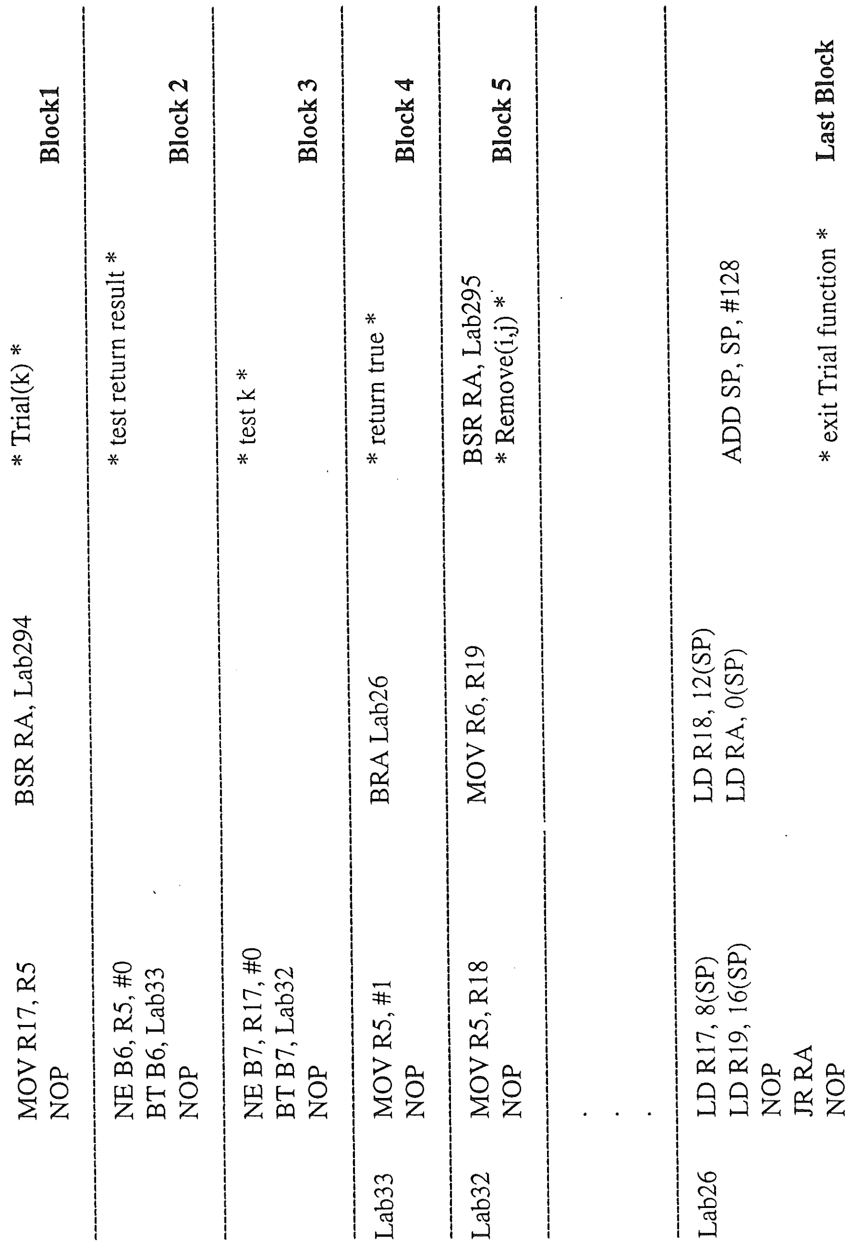



Figure 7. Locally compacted if then else statement (allowing a maximum of 2 branch, 2 memory reference and 4 ALU instructions per LIW)

| Label | Slot 1 | Slot 2 | Slot 3 | Slot 4 | Block |
| --- | --- | --- | --- | --- | --- |
| | MOV R17, R5<br>NOP | BSR RA, Lab294 | | | **Block 1** |
| | NE B6, R5, #0<br>BT B6, Lab411<br>T B6 LD R19, 16(SP) | F B6 NE B7, R17, #0<br>T B6 LD RA, 0(SP) | T B6 MOV R5, #1<br>T B6 ADD SP, SP, #128 | T B6 LD R17, 8(SP) | T B6 LD R18, 12(SP)<br>**Block 2** |
| | BT B7, Lab32<br>NOP | | | | **Block 3** |
| | MOV R5, #1<br>NOP | BRA Lab26 | | | **Compensation Code** |
| Lab32 | MOV R5, R18<br>NOP | MOV R6, R19 | BSR RA, Lab295 | | **Block 5** |
| | .<br>. | | | | |
| Lab26 | LD R17, 8(SP)<br>LD R19, 16(SP) | LD R18, 12(SP)<br>LD RA, 0(SP) | ADD SP, SP, #128 | | **Compensation Code** |
| Lab411 | NOP<br>JR RA<br>NOP | | | | **Last Block** |

Figure 8.Conditionally compacting block 2 of the if then else statement

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | MOV R17, R5<br>NOP | BSR RA, Lab294 | | | | | | **Block 1** |
| | NE B6, R5, #0<br>BT B6, Lab411<br>T B6 LD R19, 16(SP) | F B6 NE B7, R17, #0<br>T B6 LD RA, 0(SP) | T B6 MOV R5, #1<br>T B6 ADD SP, SP, #128 | T B6 LD R17, 8(SP) | T B6 LD R18, 12(SP) | | | **Block 2** |
| | BF B7, Lab411<br>F B7 LD R19, 16(SP) | F B7 MOV R5, #1<br>F B7 LD RA, 0(SP) | T B7 MOV R5, R18<br>F B7 ADD SP, SP, #128 | T B7 MOV R6, R19 | T B7 BSR RA, Lab295 | F B7 LD R17, 8(SP) | | F B7 LD R18, 12(SP)<br>**Block 3** |
| | .<br>. | | | | | | | |
| Lab26 | LD R17, 8(SP)<br>LD R19, 16(SP) | LD R18, 12(SP)<br>LD RA, 0(SP) | ADD SP, SP, #128 | | | | | **Compensation Code** |
| Lab411 | NOP<br>JR RA<br>NOP | | | | | | | **Last Block** |

Figure 9.Conditionally compacting block 3 of the if then else statement

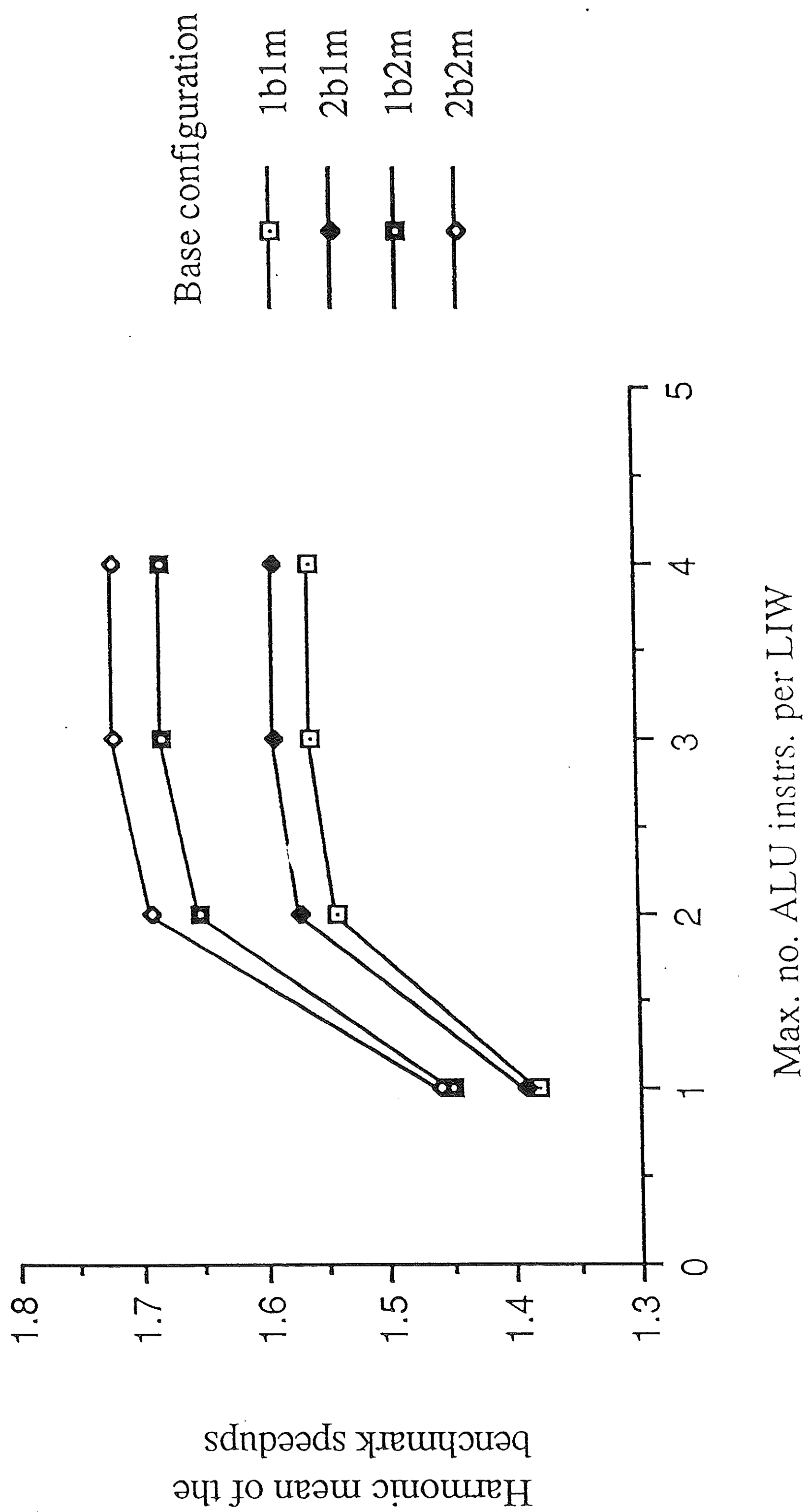



Figure 10. Average speedups for different configurations of the machine model

| Program | Description | Dynamic Instruction Count for Conditionally Compacted Sequential Code |
|---|---|---|
| Bubble | Bubble sorts 200 elements | 280063 |
| Intmm | Multiplies two 25 x 25 integer matrices | 359372 |
| Perm | Computes the permutations of 7 elements (recursive) | 393099 |
| Puzzle | Forest Baskett's program which recursively solves a cube packing problem | 41234 |
| Queens | Solves the 8 queens problem 10 times | 257526 |
| Quick | Recursive quicksort of 500 elements | 88798 |
| Towers | Solves the Towers of Hanoi for 11 discs (recursive) | 288072 |
| Tree | Performs a binary tree sort of 500 elements | 181382 |

Table 1. The Benchmark Programs

| Base Configuration | Maximum no. of branch and memory reference instructions per LIW |
|---|---|
| 1b1m | 1 branch, 1 memory reference |
| 2b1m | 2 branch, 1 memory reference |
| 1b2m | 1 branch, 2 memory reference |
| 2b2m | 2 branch, 2 memory reference |

Table 2. Base configurations of the HARP machine model

Harmonic Mean of Speedups

| | Base Configuration | | | |
|---|---|---|---|---|
| | 1b1m | 2b1m | 1b2m | 2b2m |
| non-inlined code | 1.56 | 1.59 | 1.68 | 1.72 |
| inlined code | 1.68 | 1.72 | 1.81 | 1.85 |

Table 3. Comparing the performance of conditionally compacted non-inlined and inlined sequential code for the 3 ALU variations of the four base configurations of the model

ALUsvSpeedup 1b1m Configuration

| | Prog | Cycles Seq | Cycles 1 ALU | Speedup 1 ALU | Cycles 2 ALU | Speedup 2 ALU | Cycles 3 ALU | Speedup 3 ALU | Cycles 4 ALU | Speedup 4 ALU |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | bubble | 280063 | 188522 | 1.49 | 187323 | 1.50 | 187323 | 1.50 | 187323 | 1.50 |
| 2 | intmm | 359372 | 257377 | 1.40 | 182901 | 1.96 | 181651 | 1.98 | 181601 | 1.98 |
| 3 | perm | 393099 | 266945 | 1.47 | 258271 | 1.52 | 254650 | 1.54 | 254650 | 1.54 |
| 4 | puz | 41234 | 29558 | 1.40 | 24018 | 1.72 | 22768 | 1.81 | 22753 | 1.81 |
| 5 | queens | 257526 | 176932 | 1.46 | 165992 | 1.55 | 165752 | 1.55 | 165752 | 1.55 |
| 6 | quick | 88798 | 72882 | 1.22 | 68026 | 1.31 | 66714 | 1.33 | 66714 | 1.33 |
| 7 | tower | 288072 | 218303 | 1.32 | 192676 | 1.50 | 188566 | 1.53 | 188566 | 1.53 |
| 8 | tree | 181382 | 137832 | 1.32 | 126768 | 1.43 | 126768 | 1.43 | 126768 | 1.43 |
| 9 | | | | | | | | | | |
| 10 | Harmonic Mean | | | 1.38 | | 1.54 | | 1.56 | | 1.56 |

ALUsvSpeedup 2b1m Configuration

| | Prog | Cycles Seq | Cycles 1 ALU | Speedup 1 ALU | Cycles 2 ALU | Speedup 2 ALU | Cycles 3 ALU | Speedup 3 ALU | Cycles 4 ALU | Speedup 4 ALU |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | bubble | 280063 | 188522 | 1.49 | 187323 | 1.50 | 187323 | 1.50 | 187323 | 1.50 |
| 2 | intmm | 359372 | 257377 | 1.40 | 181653 | 1.98 | 180403 | 1.99 | 180353 | 1.99 |
| 3 | perm | 393099 | 266945 | 1.47 | 251031 | 1.57 | 247408 | 1.59 | 247408 | 1.59 |
| 4 | puz | 41234 | 29264 | 1.41 | 23616 | 1.75 | 22397 | 1.84 | 22382 | 1.84 |
| 5 | queens | 257526 | 176932 | 1.46 | 163752 | 1.57 | 163512 | 1.57 | 163492 | 1.58 |
| 6 | quick | 88798 | 72882 | 1.22 | 64518 | 1.38 | 63206 | 1.40 | 63206 | 1.40 |
| 7 | tower | 288072 | 212139 | 1.36 | 186512 | 1.54 | 182402 | 1.58 | 182402 | 1.58 |
| 8 | tree | 181382 | 136578 | 1.33 | 127651 | 1.42 | 125762 | 1.44 | 125762 | 1.44 |
| 9 | | | | | | | | | | |
| 10 | Harmonic Mean | | | 1.39 | | 1.57 | | 1.59 | | 1.59 |

ALUsvSpeedup 1b2m Configuration

| | Prog | Seq | Cycles 1ALU | Speedup 1 ALU | Cycles 2 ALU | Speedup 2 ALU | Cycles 3 ALU | Speedup 3 ALU | Cycles 4 ALU | Speedup 4 ALU |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | bubble | 280063 | 188321 | 1.49 | 167221 | 1.67 | 167221 | 1.67 | 167221 | 1.67 |
| 2 | intmm | 359372 | 257366 | 1.40 | 182890 | 1.96 | 181640 | 1.98 | 181590 | 1.98 |
| 3 | perm | 393099 | 222225 | 1.77 | 213551 | 1.84 | 209930 | 1.87 | 209930 | 1.87 |
| 4 | puz | 41234 | 29478 | 1.40 | 23937 | 1.72 | 22687 | 1.82 | 22672 | 1.82 |
| 5 | queens | 257526 | 164500 | 1.57 | 153550 | 1.68 | 153310 | 1.68 | 153310 | 1.68 |
| 6 | quick | 88798 | 71496 | 1.24 | 65753 | 1.35 | 64441 | 1.38 | 64441 | 1.38 |
| 7 | tower | 288072 | 197790 | 1.46 | 170103 | 1.69 | 165993 | 1.74 | 165993 | 1.74 |
| 8 | tree | 181382 | 132087 | 1.37 | 122270 | 1.48 | 122270 | 1.48 | 122270 | 1.48 |
| 9 | | | | | | | | | | |
| 10 | Harmonic Mean | | | 1.45 | | 1.65 | | 1.68 | | 1.68 |

ALUsvSpeedup 2b2m Configuration

| | Prog | Cycles Seq | Cycles 1 ALU | Speedup 1 ALU | Cycles 2 ALU | Speedup 2 ALU | Cycles 3 ALU | Speedup 3 ALU | Cycles 4 ALU | Speedup 4 ALU |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | bubble | 280063 | 188321 | 1.49 | 167221 | 1.67 | 167221 | 1.67 | 167221 | 1.67 |
| 2 | intmm | 359372 | 257366 | 1.40 | 181642 | 1.98 | 180392 | 1.99 | 180342 | 1.99 |
| 3 | perm | 393099 | 214985 | 1.83 | 206311 | 1.91 | 202688 | 1.94 | 202688 | 1.94 |
| 4 | puz | 41234 | 29184 | 1.41 | 23535 | 1.75 | 22316 | 1.85 | 22301 | 1.85 |
| 5 | queens | 257526 | 164500 | 1.57 | 151310 | 1.70 | 151070 | 1.70 | 151050 | 1.70 |
| 6 | quick | 88798 | 71496 | 1.24 | 62245 | 1.43 | 60933 | 1.46 | 60933 | 1.46 |
| 7 | tower | 288072 | 191626 | 1.50 | 163939 | 1.76 | 159829 | 1.80 | 159829 | 1.80 |
| 8 | tree | 181382 | 130833 | 1.39 | 123153 | 1.47 | 121264 | 1.50 | 121264 | 1.50 |
| 9 | | | | | | | | | | |
| 10 | Harmonic Mean | | | 1.46 | | 1.69 | | 1.72 | | 1.72 |

Appendix 2

Performance of Inlined Code

| | Prog | Cycles Seq (non-inlined) | 1b1m Cycles 3 ALU | Configuration Speedup 3 ALU | 2b1m Cycles 3 ALU | Configuration Speedup 3 ALU | 1b2m Cycles 3 ALU | Configuration Speedup 3 ALU | 2b2m Cycles 3 ALU | Configuration Speedup 3 ALU |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | | | | | | | | | | |
| 2 | | | | | | | | | | |
| 3 | | | | | | | | | | |
| 4 | | | | | | | | | | |
| 5 | bubble | 280063 | 186311 | 1.50 | 186311 | 1.50 | 166209 | 1.69 | 166209 | 1.69 |
| 6 | intmm | 359372 | 171018 | 2.10 | 171012 | 2.10 | 171018 | 2.10 | 171012 | 2.10 |
| 7 | perm | 393099 | 204245 | 1.92 | 197007 | 2.00 | 168183 | 2.34 | 160945 | 2.44 |
| 8 | puz | 41234 | 22768 | 1.81 | 22397 | 1.84 | 22687 | 1.82 | 22316 | 1.85 |
| 9 | queens | 257526 | 165654 | 1.55 | 163414 | 1.58 | 153222 | 1.68 | 150982 | 1.71 |
| 10 | quick | 88798 | 64191 | 1.38 | 60683 | 1.46 | 61919 | 1.43 | 58411 | 1.52 |
| 11 | tower | 288072 | 145261 | 1.98 | 143213 | 2.01 | 124789 | 2.31 | 122741 | 2.35 |
| 12 | tree | 181382 | 120755 | 1.50 | 119747 | 1.51 | 116258 | 1.56 | 115250 | 1.57 |
| 13 | | | | | | | | | | |
| 14 | Harmonic Mean | | | 1.68 | | 1.72 | | 1.81 | | 1.85 |