A Register Pressure Sensitive Instruction Scheduler for Dynamic Issue Processors*

Raúl Silvera † Jian Wang ‡ Guang R. Gao §
McGill University
School of Computer Science
3480 University Street
Montreal, QC H3A 2A7. Canada

R. Govindarajan
Supercomputer Edn. & Res. Center
Dept. of Comp. Sci. & Automation
Indian Institute of Science
Bangalore, 560 012. India

Abstract

Several modern superscalar processors contain an out of order (OOO) instruction issue mechanism, which resolves dependencies between instructions to expose greater instruction level parallelism (ILP). How to extend a traditional instruction scheduler to take advantage of these hardware resources has presented both a challenge and an opportunity for compiler design.

In this paper, we present a new approach for instruction scheduling, which reorders the instructions in a traditional instruction schedule to reduce its register pressure while maintaining the amount of ILP exploitable by the target OOO processor. This may prevent the introduction of spill code, thus producing a performance improvement.

We have implemented our instruction scheduler under the MOST scheduling testbed[1]. Our experiments show that the proposed approach reduces the register pressure by 12.81% in SPEC92 *benchmark loops which do not require any spill code. For loops with a high register pressure, our approach reduced the amount of spill code required by an average of 32.08%, and produced an average performance improvement of 8.79%.*

1 Introduction

Many modern computer processors are based on an out-of-order (OOO) superscalar execution model. These processors make use of a sophisticated dynamic issue mechanism that reorders the program instructions at runtime to overcome data hazards and expose more Instruction-Level Parallelism (ILP). Some examples of commercial OOO superscalar designs include the IBM PowerPC 620[3], the MIPS R10000[14], and the Intel Pentium Pro[13].

In the OOO execution model, the hardware maintains an *instruction window* from which instructions are selected for *issue* at each clock cycle. The processor may select the instructions in an order different from the one in which they appear in the original program. The dynamic issue hardware in these processors also includes a *register renaming mechanism*[14] which maps *logical registers* (register names visible to the compiler) to *physical registers* (hardware locations in the processor) in order to remove anti- and output-dependencies and hence expose greater ILP.

Extending a traditional instruction scheduler to take advantage of the hardware resources in the presence of register renaming and OOO issuing has presented both challenges and opportunities for compiler design[9, 4]. One possible approach is to use a traditional VLIW-style instruction scheduler to derive a parallel instruction schedule. Then, generate from that schedule an instruction sequence from which the processor can exploit all the ILP that appears in the schedule. Throughout this paper, we refer to the process of generating an instruction sequence from a parallel schedule by the term *linearization.*

Even though the register renaming and OOO issue mechanisms of a processor are transparent to its instruction set, we believe that knowledge of their presence may allow the compiler to generate more efficient code. In this paper, we present a new approach for basic block instruction scheduling which is sensitive to the register renaming and OOO issue capabilities of the target processor. Our approach consists of using a traditional instruction scheduler to generate a parallel schedule, and then applying a *register-pressure sensitive* linearization algorithm to generate an instruction

*This work was supported by research grants from NSERC, Micronet – Network Centers of Excellence, Canada.

†Currently affiliated to the IBM Toronto Lab., 1150 Eglinton Avenue East 3r/362. North York, Ontario M3C 1H7. Canada

‡Currently affiliated to the Nortel Montreal Lab., 16 Place du Commerce. Nuns' Island, Quebec H3E 1H6. Canada

§Currently affiliated to the Department of Electrical and Computer Engineering, University of Delaware. 104 Evans Hall. Newark, DE 19716

sequence with a reduced register pressure. Such an approach will result in performance improvements by reducing the amount of spill code introduced by the register allocator. Since the superscalar processors depend heavily on ILP exploitation to achieve their levels of performance, we must ensure that our transformations do not reduce the degree of ILP exploitable from the program by the target processor. The relationship between the instruction scheduling and register allocation processes for multi-issue architectures has been thoroughly studied in the literature [10, 4, 2, 12]. However, to the best of our knowledge, none of these studies have considered the presence of an OOO issue mechanism in the target architecture.

We implemented our instruction scheduling algorithm in the MOST scheduling testbed[1]. Our scheduling method is used to schedule several basic block loop bodies extracted from the *SPEC92* benchmark suite. We evaluated the performance of the constructed schedules by executing the scheduled code in a cycle-by-cycle superscalar processor simulator. The instruction scheduler used to generate the initial schedules was based on Huff's slack scheduling algorithm[7]. Comparing the code generated using our technique against the one generated using a trivial linearization algorithm, we have obtained the following results. For loops that do not require the introduction of spill code, our approach reduced the register pressure of the program, on an average, by 12.81%, without degrading the execution performance. For loops with a high register pressure, our approach reduced the amount of spill code introduced, on an average, by 32.08%; this, in turn, resulted in an average performance improvement of 8.79%.

The rest of this paper is organized as follows: In Sections 2 and 3, we briefly discuss the architecture model used in our study, and present an example to illustrate the optimization opportunities that we have discovered. In Section 4, we present the problem formulation. Sections 5 and 6 deal with our proposed solutions. In Section 7, we present an example of the execution of our algorithm. In Section 8, we report our experimental results. Conclusions are stated in Section 9.

2 The Superscalar Architecture Model

This paper studies the instruction scheduling problem on dynamic-issue superscalar processors. The main objective of the dynamic issue mechanism of a superscalar processor is to expose more ILP by overcoming data hazards. The processor is equipped with an FIFO queue that serves as a window over the program sequence, and which is used to store the operands and results of the corresponding instructions. As instructions become ready to execute, they are dispatched from this window to the functional units of the processor. Completed instructions are retired/graduated in program order and leave the instruction window, making room for new instructions.

This model supports OOO issue because instructions may be dispatched out of program order to the functional units of the processor. Also, it supports register renaming, because the instruction window contains a separate location for storing the operands and results of each instruction. A more detailed description of this model can be found in [8].

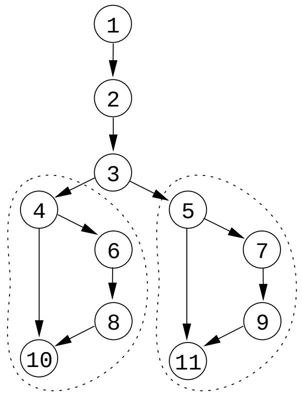
3 A Motivating Example

Consider an instruction sequence generated by a traditional ILP scheduler for a superscalar in-order processor. The introduction of an OOO issue and register renaming mechanism in the processor makes it possible for the compiler to statically rearrange the instructions in the sequence to reduce its register pressure while maintaining the amount of ILP that can be exploited by the target processor.

Let us study a sample program to illustrate this observation. Consider the DLX[6] assembly code in Figure 1(a), which was generated by the DLXCC compiler for a basic block extracted from the *039.wave5* program of the *SPEC92* benchmark suite. The DDG of this code is shown in Figure 1(b).

```
 1: add r1,r29,#-1
 2: slli r1,r1,#2
 3: add r1,r30,r1
 4: add r2,r1,r28
 5: add r1,r1,r27
 6: lf f0,0(r2)
 7: lf f2,0(r1)
 8: multf f0,f6,f0
 9: multf f2,f6,f2
10: sf 0(r2),f0
11: sf 0(r1),f2
```

a) A sample program



b) Data dependence graph

Figure 1: Example loop from the *wave5d* benchmark.

For an in-order superscalar implementation of the DLX architecture with two general-purpose functional units of unit latency, a traditional instruction scheduler could generate the parallel schedule shown in Figure 2. According to this schedule, the instructions in the two main branches of the DDG could be executed in parallel, and the program is executed in 7 clock cycles. To effectively use this schedule, the compiler must linearize this schedule into an instruction sequence that, when presented to the target processor, executes as described by the schedule. To avoid dependence stalls, the compiler ensures that all instructions scheduled at a given clock cycle are contiguous in its linearized program sequence.

Clock Cycle	Instructions
1	1
2	2
3	3
4	4 5
5	6 7
6	8 9
7	10 11

Figure 2: An example instruction schedule for the program shown in Figure 1

However, when linearizing a schedule to be executed by a dynamic issue processor, the instructions to be executed at a given clock cycle no longer have to be contiguous in the program sequence. For example, consider the assembly code in Figure 3, which was produced by reordering the instructions of our example program. A trace of the execution of this sequence on an OOO version of the DLX processor is shown in Figure 4. For simplicity, let us assume that the processor is able to fetch, execute and retire at a single clock cycle as many instructions as can fit in the lookahead window. Let us follow this trace for each clock cycle.

```
 1:   add r1,r29,#-1
 2:   slli r1,r1,#2
 3:   add r1,r30,r1
 4:   add r2,r1,r28
 6:   lf f0,0(r2)
 8:   multf f0,f6,f0
10:   sf 0(r2),f0
 5:   add r1,r1,r27
 7:   lf f0,0(r1)
 9:   multf f0,f6,f0
11:   sf 0(r1),f0
```

Figure 3: A reordering of the program in Figure 1

Cycles 1-3: In the first three clock cycles, there is only a single data-ready instruction in the lookahead window.

Cycle 4: The lookahead window covers instructions 4, 6, 8, 10 and 5. Among these, only 4 and 5 are data-ready. Since these instructions are independent from each other; they are executed in parallel by the processor.

Cycle 5: The lookahead window covers instructions 6, 8, 10 and 7. Out of these, only instructions 6 and 7 are ready; however, these two instructions modify the value of the register `f0`. Fortunately, this hazard is solved by the register renaming mechanism of the processor, as it provides separate hardware registers to store the result of each instruction in the lookahead window.

Cycle 6: The lookahead window covers instructions 8, 10 and 9. Out of these, only 8 and 9 are ready. The register renaming mechanism ensures that these instruction read the corresponding value of the `f0` register, and provides with separate hardware registers to store their results.

Cycle 7: The lookahead window covers instructions 10 and 11. These two instructions are data-ready; the register renaming mechanism ensures that each instruction reads the corresponding value of the `f0` register.

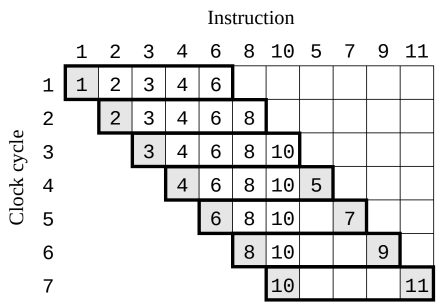


Figure 4: Execution trace of the reordered code on an OOO superscalar processor with a lookahead window of size 5. Each row of this diagram represents the processor state at each clock cycle: the thick frame represents the lookahead window, and the instructions selected for issue appear in dark-shaded boxes.

We can see that this sequence is executed exactly as described by the schedule in Figure 2, even though the order appears to be counter-intuitive in terms of avoiding data-dependency stalls. The reason for this is that the processor has enough lookahead to discover independent instructions and execute them in parallel. An in-order version of the processor would not be able to exploit that parallelism, so it would have achieved a lower performance compared to the original code.

By carefully studying the two sequences, we can observe an advantage of the reordered sequence: it has a lower register pressure. In the original sequence, the live ranges of the values produced by instructions 6 and 7 are overlapped. So, they must be assigned separate registers by the register allocator. In contrast, those live ranges do not overlap in the reordered sequence, so they can be assigned to the same logical register. This does not produce issue stalls in the processor, as the register renaming mechanism of the OOO processor provides different physical locations for each of these values and allows the two branches of the DDG to be executed in parallel.

For loops having a high register pressure, reordering the instructions in a program may reduce the amount of spill code introduced by the register allocator, thus improving the performance of the program. However, this reordering must be performed carefully, to ensure that the OOO processor will be able to exploit from the reordered sequence as much ILP as present in the original sequence.

4 Problem Formulation

In the previous section we discussed the feasibility of reordering the instructions of a program to reduce its register pressure while maintaining the amount of ILP exploitable by a given OOO superscalar processor. In this study, we propose an instruction scheduling technique that takes advantage of this possibility. Our approach uses a *register-sensitive linearization algorithm* to generate an instruction sequence for a schedule Σ produced by a traditional VLIW-style scheduler. Our goal is to generate a low register pressure sequence from which the target OOO superscalar processor can exploit as much ILP as from the sequence trivially generated from Σ. More formally, we can define this problem as:

> *Given a program, a dynamic issue superscalar processor M, and an instruction schedule Σ, generate a linearized register-sensitive sequence (LRS) from which the processor M can exploit as much ILP as present in Σ, and has a lower register pressure than the instruction sequence trivially derived from Σ.*

In the remainder of this paper, we denote Σ_c as the set of instructions that appear in Σ at the clock cycle c, and $\Sigma_{[a \cdots b)}$ as the set of instructions on Σ between cycles a and $b-1$, both inclusive.

5 The Matching Conditions

As previously mentioned, when linearizing a parallel schedule, it is important not to degrade the ILP that can be exploited by the target processor. The general problem is how to predict, at compile time, which instructions the processor will select for issue at each clock cycle from a given program sequence. We conjecture that this is a difficult problem; so, we propose to use an approximate approach to address it.

We define a set of conditions over an instruction sequence that are necessary for an OOO processor to exploit all the ILP present in a given parallel schedule. We have named these conditions as the *Matching Conditions*. Further, if an instruction sequence satisfies all these conditions, we say that it *matches*, in terms of the level of ILP exploited, the given schedule. Last, if an instruction sequence matches a given schedule, we assume that it is likely that the target OOO processor can exploit from that sequence as much ILP as is present in the parallel schedule. We expect to evidence the accuracy of this assumption in our experimental results.

5.1 The First Matching Condition

Let us consider an instruction schedule Σ, and an instruction $s \in \Sigma_c$ with a latency of l clock cycles. We can divide the instructions in Σ into four groups, according to their relation to s. Group I is formed by instructions scheduled to start their execution before s. Group II consists of instructions scheduled to start simultaneously with s. Group III is formed by instructions scheduled to start their execution after s has started, but before scheduled to start their execution after s has completed its execution. Figure 5 contains a diagram describing these four groups.

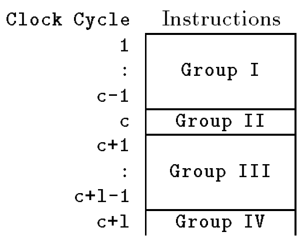


Figure 5: Classification of the instructions in a schedule Σ according to their relation to an instruction $s \in \Sigma_c$ of latency l

Group I. This group consists of instructions in $\Sigma_{[1 \cdots c)}$. According to the schedule, these instructions must start execution before s does. We must ensure that, after reordering, it is possible for the processor to start the execution of all instructions in this group before s starts execution.

Consider an instruction s_1 in this group. If this instruction appears in an LRS after instruction s, it may be possible for s to start execution before s_1 does. In particular, if s appears in the LRS at least W instructions earlier than s_1, instruction s_1 would not be able to enter the lookahead window until the instruction s completes execution and leaves the window. So, the instruction s would be forced to start execution before s_1 does.

To illustrate this point, consider the program in Figure 1, and an OOO superscalar implementation of the DLX architecture with a lookahead window of length 4, able to execute any pair of instructions at a single clock cycle. Can this processor realize the schedule shown in Figure 6 from the sequence `1-2-3-5-7-9-11-4-6-8-10`?

Clock Cycle	Instructions
1	1
2	2
3	3
4	4
5	6 5
6	8 7
7	10 9
8	11

Figure 6: Example ILP schedule

The answer to this question is no. According to the target schedule, instruction 4 must start execution before instruction 5. However, since instructions enter the lookahead window in program order, instruction 5 will enter the window before 4 does. For the instruction 4 to start execution before 5, instruction 5 would have to stay in the window until 4 starts execution. However, these two instructions cannot appear simultaneously in the lookahead window, as they are separated in the program by a distance larger than the size of the window. So, it is impossible for these two instructions to start execution in the order in which they appear in the schedule.

Note that this is not the case for instructions 7 and 4, for example; it may be possible for 4 to start execution before 7, even though 7 appears at an earlier position in the instruction sequence. This is because they are separated by a distance of 4, so it is possible for instruction 4 to enter the lookahead window and start execution while 7 is still in the window.

From this analysis, we can conclude that all instructions in Group I must appear in the LRS either before instruction s or no later than W instructions after it. That is, if s appears in the LRS at index i, all instructions in $\Sigma_{[1\cdots c)}$ must appear before index $i+W$.

Group II. This group is formed by the instructions in Σ_c. All instructions in this group must start execution at the clock cycle c. For this to happen, they must all be present in the lookahead window at that cycle. So, they must be separated in the program sequence by a distance less than or equal to the size of the instruction window. In other words, all instructions in Σ_c must appear in the LRS before index $i+W$.

Group III. This group is formed by instructions in $\Sigma_{[c+1\cdots c+l)}$. These instructions must start execution after s does, but before the execution of s is completed. We must guarantee that all these instructions can enter the lookahead window before s completes, so that they can start execution at their scheduled cycles.

Consider an instruction $s_3 \in \Sigma_{c+m}$ (with $1 \leq m < l$). If s appears in the LRS at index i and s_3 appears in the LRS at or after index $i+W$, then s_3 will not enter the lookahead window until s completes execution. This is not the desired behavior, as s will complete execution at cycle $c+l$, and according to the schedule, s_3 must start execution before that cycle.

Let us illustrate this point by using an example. Consider the program in Figure 1, and an OOO superscalar architecture that can start at each clock cycle the execution of two instructions of different type. Assume that all instructions have a unit latency, except for instructions 8 and 9, which have a latency of 2 clock cycles. Can this processor exploit from the instruction sequence `1-2-3-4-6-8-10-5-7-9-11` all the ILP present in the schedule of Figure 7?

To answer this question, let us study the execution trace of this sequence, shown in Figure 7. Instruction 9 is scheduled to start execution at the clock cycle 7; however, by looking at the execution trace we can see that it cannot start execution until cycle 8. The reason for this is that the instruction 9 is unable to enter the lookahead window at cycle 7, because instruction 8, which has a latency of two clock cycles, prevents the lookahead window from advancing at clock cycle 7. For instruction 9 to start execution at its corresponding clock cycle, it would have to enter the lookahead window before instruction 8 completes execution. This is only possible if 9 appears in the instruction sequence either before instruction 8, or no later than W instructions after it.

From this analysis we can conclude that all instructions in Group III must appear in the LRS either before instruction s or no later than W instructions after it. That is, if s appears in the LRS at index i,

Clock Cycle	Instructions
1	1
2	2
3	3
4	4
5	6 5
6	8 7
7	9
8	10
9	11

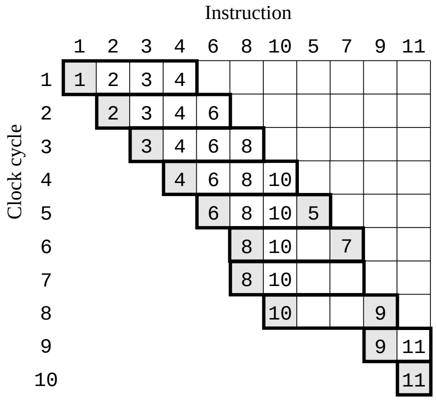


Figure 7: An ILP schedule, and the execution trace of the code generated by the linearization algorithm

all instructions in $\Sigma_{[c+1\cdots c+l)}$ must appear in the *LRS* before index $i+W$.

The first matching condition. The constraints described in the three previous sections indicate that if s is to be placed at index i in the sequence, all of the instructions in Groups I, II and III must appear in the program sequence before index $i+W$. We have defined a simple test to evaluate this condition, which we have named the *First Matching Condition*. This test is defined as follows.

> *If an instruction $s \in \Sigma_c$ appears in the LRS at index i, all instructions in $\Sigma_{[1\cdots c+l)}$ must appear in the LRS before index $i+W$.*

5.2 The Second Matching Condition

This condition deals with the resource constraints of the processor. At each clock cycle, the dynamic issue mechanism of the processor selects several instructions from the lookahead window and starts their execution. If the processor does not have enough hardware resources to start the execution of all ready instructions in the lookahead window, it must decide which instructions to start at the current cycle, and which will have to wait in the lookahead window. If we assume that the processor prioritizes between the ready instructions in the lookahead window by using their order in the program sequence, there may exist some situations in which this policy forces the processor to start the execution of a pair of instructions in a particular order.

Consider two instructions $s_1 \in \Sigma_c$ and $s_2 \in \Sigma_d$, with $c < d$. If s_2 appears earlier than s_1 in the program sequence, the processor will give priority to s_2 over s_1 when selecting the instructions to start execution at cycle c. Since the instruction s_1 appears before s_2 in Σ, we must ensure that the processor is able to skip instruction s_2 and start first the execution of s_1. To make it possible for the processor to skip s_2 and start first the execution of instruction s_1, we need one of the following cases to apply:

- The two instructions are executed by different functional units. In this case, it may be possible that there is no functional unit available to execute instruction s_2 at clock cycle c. So, s_1 may be selected by the processor at cycle c, before s_2.
- Instruction s_2 is not data ready at the clock cycle c. Since instruction s_2 cannot be executed in the clock cycle c because some of its operands are unavailable, it is skipped by the dynamic issue mechanism of the processor, thus allowing the instruction s_1 to start execution as scheduled at cycle c.

In any other case, the processor will start the execution of the instruction s_2 before instruction s_1. To prevent this from happening, we have defined a test to determine whether one of these two cases applies. We call this test the *Second Matching Condition*, which is defined as follows.

> *If two instructions $s_1 \in \Sigma_c$ and $s_2 \in \Sigma_{c+k}$ utilize the same type of functional unit, with $k \geq 1$, and all the data dependencies of s_2 are satisfied at cycle c, then s_1 must appear in the LRS at an earlier position than s_2.*

6 The Linearization Algorithm

We have developed an algorithm to linearize a parallel instruction schedule into a program sequence that matches the schedule. It receives as inputs an instruction schedule Σ for a program and the architectural parameters of the processor, and produces a *Linearized Register-Sensitive Sequence* (*LRS*) that matches the given schedule for that particular processor.

Our algorithm is an extension of the list scheduling algorithm. It walks over the DDG of the program, selecting at each step one instruction to be appended to the current instruction sequence. At each step of the algorithm we apply several tests to the list of data-ready instructions, to disqualify any instruction whose placement at the current position would prevent the final sequence from matching the target schedule. If there are several instructions that pass these tests, one of them is selected by using certain heuristics.

6.1 The Matching Checks

At each iteration of the linearization algorithm, we verify each instruction in the data-ready list to determine if they can be placed in the current position of the sequence without risk of preventing the final sequence from matching the schedule. We call this set of tests, the *matching checks*; in this section, we explain how these checks are performed.

Let us assume that at the iteration i of the linearization algorithm, instruction $s \in \Sigma_c$ is present in the data ready list, so it is being considered for placement at index i in the LRS, and that all instructions that produce values used by s are scheduled to complete execution at cycle $c' \leq c$. We have defined two tests to ensure that the instruction sequence being generated will satisfy the two matching conditions. The first test, which we call the "can" test, verifies whether the candidate instruction s can be selected at the current iteration of the algorithm without breaking any of the two conditions. The second test, which we call the "must" test, indicates whether the instruction to be placed at the current position of the sequence must be selected from a restricted subset of the data-ready list. This test is defined to ensure that the result sequence will honor the constraints introduced by the first matching condition in previous steps of the algorithm.

The "can" Test. According to the first matching condition, if s is placed at index i in the LRS, all instructions in $\Sigma_{[1\cdots c+l)}$ not yet placed in the sequence must be placed in the following $W-1$ positions of the sequence. This is clearly impossible if the number of instructions in $\Sigma_{[1\cdots c+l)}$ not yet placed in the sequence is larger than $W-1$, as there would not be enough space in the next $W-1$ places of the sequence to accommodate all those instructions. According to the second matching condition, s can only be selected if all instructions in $\Sigma_{[c'\cdots c)}$ that use the same functional unit as s have already been placed in the sequence.

So, for each instruction s in the data-ready list, we verify whether the number of instructions in $\Sigma_{[1\cdots c+l)}$ not yet placed in the sequence is less than W, and whether all the instructions in $\Sigma_{[c'\cdots c)}$ that use the same functional unit as s have already been placed in the sequence. Only those instructions that pass this test will be considered for placement at the current position of the instruction sequence.

The "must" Test. This test is used to enforce the constraints introduced by the first matching condition in previous steps of the algorithm, as it indicates that the instructions in $\Sigma_{[1\cdots c+l)}$ not yet placed in the sequence must be selected during the following $W-1$ iterations of the algorithm. To enforce this rule, we will associate a number named $last$ to each instruction in the program, which is initialized as ∞. At each iteration i of the linearization algorithm, after the algorithm selects instruction s for placement at the current position of the sequence, the value of $last$ associated to each instruction $s' \in \Sigma_{[1\cdots c+l)}$ not yet placed in the sequence is set to $min(last(s'), i+W-1)$. This indicates that those instructions must be placed in the sequence before index $i+W$, as required by the first matching condition.

The algorithm must guarantee that instructions will be placed in the sequence at or before their value of $last$. At iteration i, the algorithm determines whether there exist n instructions not yet placed in the sequence with a value of $last$ equal to $i+n$. If so, it must select one of those instructions at the current iteration, or else it will be impossible to accommodate those n instructions in the next $n-1$ positions of the sequence.

6.2 The Linearization Algorithm

A high-level description of our algorithm is shown in Figure 8. The `initReady` and `updateReady` procedures are taken from the original list scheduling algorithm; they manage the *ready* variable, which contains the list of data-ready instructions at each iteration of the program.

At the beginning of each iteration of the algorithm, we apply the "can" test to determine which instructions from the data-ready list can be placed at the current position in the sequence. This test is implemented by the `canTest` procedure, which checks each instruction in the ready list, and verifies the test described in Section 6.1. If an instruction does not pass this test, it is not considered for placement at the current position in the sequence. After applying the "can" test, we use the `mustTest` function to determine whether there exists a set of n instructions with a value of $last$ less than or equal to $i+n$. If so, the algorithm uses the `mustTestList` procedure to obtain that set, and

```
Inputs:
    DDG:  The DDG of the given program
    Σ:    The schedule of instructions to match
    W:    The size of the lookahead window of the processor
    N:    The number of instructions in the program
Output:
    An instruction sequence LRS that matches Σ
Body:
 LRS = ∅
 foreach s ∈ S                ;Initialize the value of L
   L(s) = ∞
 R = initReady(DDG,Σ)         ;Find the initial data-ready list
 while R ≠ ∅                  ;While R is not empty
   R' = canTest(R,Σ,W)        ;Apply the can test
   if (mustTest(i,W,L))       ;Apply the must test
     R' = R' ⋂ mustTestList(i,W,L)
   s = heuristic(R')          ;Select one instruction
   LRS = LRS + s              ;Append s to the output sequence
   L = updateLast(s,Σ,W)      ;Add constraints for the must test
   R = updateReady(s,Σ)       ;Update the data ready list
 return LRS
```

Figure 8: The Linearization Algorithm

intersects it with the set of instructions that passed the "can" test to obtain the actual set from which the next instruction will be selected; otherwise, it uses the set produced by the "can" test. The algorithm selects one instruction from those that passed the two tests by using certain heuristics. The *last* data structure is maintained by the `updateLast` procedure. Whenever an instruction s is placed in the sequence, this procedure updates the value of *last* of all instructions scheduled to start execution before s completes. The *last* value of those instructions is updated to $i+W-1$, to indicate that those instructions must be placed in the sequence before index $i+W-1$. Note that the value of *last* for any given instruction needs to be updated at most once during the whole algorithm.

6.3 The Register Pressure Heuristics

The linearization algorithm uses heuristics to select the instruction to place at each position of the program sequence. In this section, we describe a set of heuristics aimed at reducing the overall register pressure of the resulting sequence. Many compiler optimizations for exposing ILP overlap the live ranges of the program variables, which tends to increase the register pressure of the program. Our heuristics aim at reverting this effect, by reducing the overlap of the live ranges of the variables.

Our heuristics are based on a priority function, which assigns a priority value to each instruction being considered for placement at the current position of the sequence. The instruction with the larger priority is selected for placement by the heuristic; ties are solved by using the order of the instructions in the input schedule. Our priority function, shown in

```
Inputs:
    i:     The corresponding instruction
    DDG:   The DDG of the given program
Output:
    An integer value that represents the heuristic
Body:
  priority = 0
  foreach instruction j on which i depends
    if all other dependents of j have been selected already,
      priority=priority+2
    else
      priority=priority+1
  foreach instruction j that depends on i
    if j depends on other instructions not yet selected.
      priority=priority-1
    else
      priority=priority-2
  return priority
```

Figure 9: The priority function used by the heuristics

Figure 9, tries to reduce the length of the live ranges of the variables, hoping to reduce the overlap between them. This function calculates the priority value for an instruction by using the number of input operands, the number of instructions that use the value it produces, and whether those instructions have additional operands that have not yet been produced.

The first loop in our priority function verifies each operand of the candidate instruction i. If all the other instructions that use that value have already been placed in the sequence, the selection of instruction i would free the register occupied by that value. So, in this case, the algorithm increases the priority of instruction i by 2. Otherwise, the selection of instruction i would not immediately free a register, but it would be a step in freeing the register occupied by the respective operand. In this case, the algorithm increases the priority of instruction i by 1.

The second loop in the priority function examines the instructions that use the value produced by the candidate instruction i. If an instruction dependent on i is also dependent on other instruction not yet placed in the sequence, the live range of the value produced by i is expected to be long, so the priority of i is decreased by 2. Also, if the number of instructions dependent on the value produced by i is large, the live range of i is expected to be long. So, the instruction i is penalized by 1 for each dependent instruction.

7 An Execution Example

Let us present the linearization of a sample program to illustrate our algorithm. Given the program in Figure 1, let us apply the linearization algorithm to the schedule shown in Figure 7, for an OOO superscalar processor capable of starting simultaneously the execution of two instructions of different types. Assume that all instructions have a unit latency, except for instructions 8 and 9, that have a latency of 2 clock cycles. For simplicity, in this example we will use a heuristic that gives priority to instructions in the left branch of the DDG of the program.

A trace of the execution of the algorithm is shown in Figure 10. Each row in that table represents an iteration of the linearization algorithm; each column represents the instructions in the data structures used. A brief description of each column is presented below.

R: The list of data ready instructions.
Can: The instructions in **R** that pass the "can" test.
Must: The instructions that pass the "must" test.
R': If **Must** in non-empty, this row contains the intersection of **Must** with **Can**; otherwise, it is a copy of **Can**.
Selected: This is the instruction selected by our heuristic from **R'**.
Constr.: The list of instructions that must be scheduled in the next $W-1$ iterations of the algorithm.

Step	R	Can	Must	R'	Select	Constr.
1	1	1		1	1	
2	2	2		2	2	
3	3	3		3	3	
4	4 5	4		4	4	
5	5 6	5 6		5 6	6	5
6	5 8	5 8		5 8	8	7 9
7	5 10	5 10	5 7 9	5	5	
8	7 10	7 10	7 9	7	7	
9	9 10	9 10	9	9	9	10
10	10 11	10 11		10 11	10	
11	11	11		11	11	

Figure 10: Execution trace of the algorithm

In the first three iterations of the algorithm there is only one instruction in **R**; those instructions are selected in turn for placement at the sequence. In the fourth iteration, there are two ready instructions: 4 and 5. Note that the only value used by the instruction 5 is produced by instruction 3, which must have completed execution at the cycle in which instruction 4 is scheduled. Since instructions 4 and 5 use the same functional unit, and instruction 4 is scheduled to start at an earlier cycle than instruction 5, instruction 5 does not pass the "can" test. Only instruction 4 is present in the final list **R'**', and it is selected for placement at the current position in the sequence.

In the fifth iteration, there are two instructions in **R**: 5 and 6. They both pass the "can" test, and our heuristic selects 6. Since instruction 5 is scheduled to start execution at the same cycle as 6, we must place a constraint over 5 to indicate that it must be selected at or before the iteration $i+W-1=5+4-1=8$.

In the sixth iteration, there are two instructions in **R**: 5 and 8. They both pass the "can" test, and our heuristic chooses the instruction 8. In this case, we must place a constraint over instruction 7, because it is scheduled at the same cycle as the instruction 8, and over instruction 9, because it is scheduled to start execution before instruction 8 is completed (the latency of instruction 8 is 2). The constraint placed on these instructions is that they must be scheduled at or before iteration 9. Note that we should also place a constraint over instruction 5, since it is scheduled at an earlier cycle than instruction 8. However, it already has a stronger constraint, so we do not need to update it.

In the seventh iteration, there are two instructions in **R**: 5 and 10. They both pass the "can" test; however, the "must" test produces a set of instructions from which the next instruction must be selected. The reason for this is that instructions 5, 7 and 9 have been constrained to be selected at or before the ninth iteration. Intersecting this list with the list of ready instructions, we obtain a single instruction: 5, which must be selected in this iteration.

In the eighth and ninth iteration, the "must" test forces the selection of instructions 7 and 9, respectively. The selection of instruction 9 creates a constraint on instruction 10, since it is scheduled to start execution before instruction 9 completes.

The final instruction sequence produced by the algorithm is `1-2-3-4-6-8-5-7-9-10-11`. By looking at the execution trace of this sequence in the target superscalar processor, in Figure 11, we can notice that with the aid of the dynamic issue mechanism of the processor, this sequence is executed exactly as described by the given instruction schedule.

8 Experimental Results

We have designed experiments to evaluate the effectiveness of our instruction scheduling framework. Using our scheduling technique to generate code for programs with a low register pressure (which do not require of the introduction of spill code), we expect to verify that our transformations do not diminish the amount of ILP that can be exploited by the target OOO processor. Using our technique on programs with a high register pressure, we expect to observe a performance improvement, by means of a reduction in the amount of spill code introduced by the register allocator.

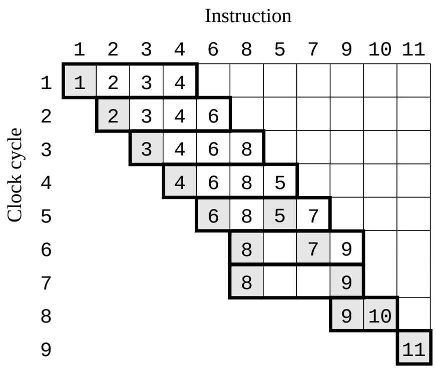


Figure 11: Execution trace of the sequence generated by the linearization algorithm

We have chosen for our experiments several basic block loops from the *mdljsp2*, *mdljdp2* and *wave5d* benchmarks of the *SPEC92* benchmark suite. We used a software pipeliner based on Huff's slack scheduling algorithm[7] to schedule the body of these loops. We applied either our register-sensitive linearization algorithm or the trivial linearization algorithm on the software pipelined scheduled to produce an instruction sequence. Finally, we ran the generated code on a processor simulator to evaluate its execution performance.

8.1 The Experimental Testbed

We have integrated several software tools to create an instruction scheduling testbed that can be used to obtain execution statistics for benchmarks written in a high level language. The major components of this testbed are:

DLXCC: This is a targeting of the gcc compiler to the DLX architecture. It was used to generate the initial assembly code for our benchmarks.

MOST: This is an instruction scheduling testbed[1]. It was used to generate a parallel schedule for each benchmark. We have integrated our linearization algorithm into this program, and included support for register allocation (based on the algorithm described in [5]), modulo code expansion and assembly code generation.

SuperDLX: This is a step-by-step simulator of an OOO superscalar implementation of the DLX architecture[11]. We use this to evaluate the performance of the code generated in our experiments. This simulator assumes a perfect cache, with a hit rate of 100%.

For our experiments, we used a configuration of the simulator with a fetch/decode/graduation rate of 4 and a lookahead window of size 32. The number and latency of the functional units of the processor is shown in Figure 12.

FU	No.	Latency	FU	No.	Latency
Int. ALU	2	1	FP. Add	1	2
Address	1	3	FP. Mul	1	8

Figure 12: SuperDLX configuration

8.2 Loops with Low Register Pressure

First, we present our experimental results for loops that do not require any spill code. For these loops, we expect the use of our linearization algorithm to reduce the register pressure of the program, while achieving the same performance as the original loops. In Table 1 we show the results of this experiment; for each loop, we show its register pressure and the number of execution cycles in the SuperDLX simulator. This table compares the results obtained by the trivial linearization algorithm with the register-sensitive linearization algorithm.

	Trivial			Reg. Sensitive		
	Execution	Registers		Execution	Registers	
Loop	Time	int	fp	Time	int	fp
mdljdp2-1	108059	7	16	108065	6	10
mdljdp2-2	96112	4	26	96064	5	20
mdljdp2-3	108060	8	10	108033	6	4
mdljdp2-4	108067	8	16	108043	5	8
mdljsp2-1	108060	8	27	108033	6	27
mdljsp2-2	108064	8	25	108043	5	25
wave5d-1	144081	11	27	144081	13	27
wave5d-2	128073	8	25	128073	8	25
wave5d-3	320177	15	27	316175	13	27
wave5d-4	128073	6	25	128073	6	25
wave5d-5	128077	6	23	128077	5	23
wave5d-6	48031	7	27	48031	6	27
wave5d-7	352717	16	23	352718	15	23
wave5d-8	64044	7	25	64044	5	25
wave5d-9	64048	7	25	64043	6	25
wave5d-10	192055	13	27	192055	11	27

Table 1: Performance comparison for low register pressure loops, with different linearization algorithms

From this table we can see that in many cases our instruction scheduling framework has been able to reduce the register pressure of the program. It can also be seen from this table that the application of our instruction scheduling framework has not degraded the performance of these programs. This indicates that our algorithm has not reduced the amount of ILP exploited by the target processor.

Loop	Execution Time	Registers	
		int	fp
mdljdp2-1	-0.01%	14.29%	37.50%
mdljdp2-2	0.05%	-25%	23.08%
mdljdp2-3	0.02%	25%	60%
mdljdp2-4	0.02%	37.50%	50%
mdljsp2-1	0.02%	25%	0
mdljsp2-2	0.02%	37.50%	0
wave5d-1	0	-18.18%	0
wave5d-2	0	0	0
wave5d-3	1.25%	13.33%	0
wave5d-4	0	0	0
wave5d-5	0	16.67%	0
wave5d-6	0	14.29%	0
wave5d-7	0	6.25%	0
wave5d-8	0	28.57%	0
wave5d-9	0.01%	14.29%	0
wave5d-10	0	15.38%	0
Average	0.09%	12.81%	10.66%

Table 2: Performance improvement for low register pressure loops with different linearization algorithms

From Table 2 we can see that our framework has reduced the integer register pressure by 12.81% on average, and by 10.66% for floating point registers. Also, we can note that our framework has degraded the performance of only one of the benchmarks (the mdljdp2-1 loop). However, in this case, the degradation was minor, and can be attributed to perturbations introduced by the dynamic nature of the issue mechanism of the processor.

8.3 Loops with High Register Pressure

Next we present the experimental results for loops that required the introduction of spill code. Table 3 presents the results for this experiment; for each loop, we show the number of variables spilled by the register allocator and the number of execution cycles required by the SuperDLX simulator.

From Table 4, we can see that the application of our instruction scheduling framework to these program has significantly reduced the amount of spill code introduced by the register allocator. This has resulted in a performance improvements of up to 27.7%, in the case of loops with a very high register pressure, such as *wave5d-4*, *mdljsp2-4* and *mdljdp2-5*. The dynamic issue mechanism of the processor is sometimes able to hide the penalties associated to spilling, so the impact of our transformation on the performance of the program varies greatly among the benchmarks used in our experiments.

Loop	Trivial			Reg. Sensitive		
	Execution Time	Spills		Execution Time	Spills	
		int	fp		int	fp
mdljdp2-1	3140804	17	76	2928751	18	65
mdljdp2-2	7169805	49	180	6597667	48	158
mdljdp2-3	1128288	0	20	960245	0	13
mdljdp2-4	3848986	17	83	3708943	16	75
mdljdp2-5	6937738	19	141	6037513	17	106
mdljdp2-6	664170	0	3	532138	0	0
mdljsp2-1	1744451	17	22	1596412	18	14
mdljsp2-2	5629423	47	115	5333349	49	97
mdljsp2-3	2660677	20	44	2464628	20	33
mdljsp2-4	828422	2	8	676179	0	0
wave5d-1	728410	36	0	692406	36	0
wave5d-2	300197	42	0	288198	42	0
wave5d-3	428928	108	0	412897	96	0
wave5d-4	360182	5	0	260166	1	0
wave5d-5	1256344	0	11	1152336	0	9
wave5d-6	1800906	68	0	1512766	44	0
wave5d-7	3168797	50	0	2952743	41	0
wave5d-8	1560395	7	0	1452369	5	0
wave5d-9	2864726	13	21	2776711	13	16
wave5d-10	2984751	14	61	2800705	13	51
wave5d-11	1080562	4	4	1014530	4	2
wave5d-12	2864726	13	21	2776711	13	16
wave5d-13	1700919	40	24	1632887	36	24
wave5d-14	1280330	2	10	1152299	3	6
wave5d-15	616328	54	0	595666	53	0

Table 3: Performance comparison for high register pressure loops with different linearization algorithms

9 Conclusions

In this paper, we have studied the instruction scheduling problem for dynamic issue processors. We have discovered certain optimization opportunities that can be exploited by an instruction scheduler to improve the performance of its generated code. These opportunities are related to the possibility of using the OOO execution and register renaming mechanisms of the processor to reduce the register pressure of a program sequence without decreasing the amount of ILP that the processor can exploit from it. This translates into a performance improvement for programs with a high register pressure by reducing the amount of spill code that must be introduced by the register allocator.

We have developed a heuristic instruction scheduling algorithm to take advantage of these opportunities. We have implemented this algorithm and integrated it into the MOST instruction scheduling testbed developed at McGill University. Using this implementation, we have performed experiments to determine the effectiveness of our approach.

Our experimental results confirm that the proposed instruction scheduling algorithm is able to improve the performance of loops having a high register pressure by 8.79% on the average and up to 27% in certain cases. For loops with a low register pressure, that do not require the insertion of spill code, our framework has been able to reduce the register pressure of the programs, by 12.81% on the average, without degrading their execution performance.

References

[1] E. R. Altman. *Optimal Software Pipelining with*

Loop	Execution Time	Spills	
		int	fp
mdljdp2-1	6.75%	-5.88%	14.47%
mdljdp2-2	7.98%	2.04%	12.22%
mdljdp2-3	14.89%	N/A	35%
mdljdp2-4	3.64%	5.88%	9.64%
mdljdp2-5	12.98%	10.53%	24.82%
mdljdp2-6	19.88%	N/A	100%
mdljsp2-1	8.49%	-5.88%	36.36%
mdljsp2-2	5.26%	-4.26%	15.65%
mdljsp2-3	7.37%	0	25%
mdljsp2-4	18.38%	100%	100%
wave5d-1	4.94%	0	N/A
wave5d-2	4%	0	N/A
wave5d-3	3.74%	11.11%	N/A
wave5d-4	27.77%	80%	N/A
wave5d-5	8.28%	N/A	18.18%
wave5d-6	16%	35.29%	N/A
wave5d-7	6.82%	18%	N/A
wave5d-8	6.92%	28.57%	N/A
wave5d-9	3.07%	0	23.81%
wave5d-10	6.17%	7.14%	16.39%
wave5d-11	6.11%	0	50%
wave5d-12	3.07%	0	23.81%
wave5d-13	4%	10%	0
wave5d-14	10%	-50%	40%
wave5d-15	3.35%	1.85%	N/A
Average	8.79%	11.11%	32.08%

Table 4: Performance improvement for high register pressure loops with different linearization algorithms

Function Unit and Register Constraints. PhD thesis, McGill University, Montréal, Québec, October 1995.

[2] D. A. Berson, R. Gupta, and M. L. Soffa. Resource spackling: A framework for integrating register allocation in local and global schedulers. In M. Cosnard, G. R. Gao, and G. M. Silberman, editors, *Proceedings of the IFIP WG 10.3 Working Conference on Parallel Architectures and Compilation Techniques, PACT '94*, pages 135–145, Montréal, Québec, August 24–26, 1994. North-Holland Publishing Company.

[3] K. Diefendorff, R. Oehler, and R. Hochsprung. Evolution of the PowerPC architecture. *IEEE Micro*, 14(2):34–49, April 1994.

[4] K. Ebcioglu, R. D. Groves, K.-C. Kim, G. M. Silberman, and I. Ziv. VLIW compilation techniques in a superscalar environment. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 36–48, Orlando, Florida, June 20–24, 1994. *SIGPLAN Notices,* 29(6), June 1994.

[5] L. J. Hendren, G. R. Gao, E. R. Altman, and C. Mukerji. A register allocation framework based on hierarchical cyclic interval graphs. In U. Kastens and P. Pfahler, editors, *Proceedings of the 4th International Conference on Compiler Construction, CC '92*, number 641 in Lecture Notes in Computer Science, pages 176–191, Paderborn, Germany, October 5–7, 1992. Springer-Verlag.

[6] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., San Francisco, California, 2nd edition, 1996.

[7] R. A. Huff. Lifetime-sensitive modulo scheduling. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 258–267, Albuquerque, New Mexico, June 23–25, 1993. *SIGPLAN Notices,* 28(6), June 1993.

[8] M. Johnson. *Superscalar Microprocessor Design.* Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1991.

[9] M. S. Lam. Instruction scheduling for superscalar architectures. *Annual Review of Computer Science*, 4:173–201, 1990.

[10] S.-M. Moon and K. Ebcioğlu. An efficient resource-constrained global scheduling technique for superscalar and VLIW processors. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 55–71, Portland, Oregon, December 1–4, 1992. ACM SIGMICRO and IEEE-CS TC-MICRO. *SIG MICRO Newsletter* 23(1–2), December 1992.

[11] C. Moura. SuperDLX — a generic superscalar simulator. ACAPS Technical Memo 64, School of Computer Science, McGill University, Montréal, Québec, April 1993.

[12] C. Norris and L. L. Pollock. An experimental study of several cooperative register allocation and instruction scheduling strategies. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 169–179, Ann Arbor, Michigan, November 29–December1, 1995. IEEE-CS TC-MICRO and ACM SIGMICRO.

[13] D. B. Papworth. Tuning the Pentium Pro microarchitecture. *IEEE Micro*, 16(2):8–15, April 1996.

[14] K. C. Yeager. The Mips R10000 superscalar microprocessor. *IEEE Micro*, 16(2):28–40, April 1996.