JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 16, 1-23 (2000)
DOI:10.1688/JISE.2000.16.1.1

A Compiler-Based Speculative Execution Scheme For ILP Enhancement

L. WANG AND TED C. YANG+
Graduate Institute of Information Engineering
Feng Chia University
Taichung, Taiwan 407, R.O.C.
+*Computer & Communications Research Lab.*
Industrial Technology Research Institute
Hsinchu, Taiwan 310, R.O.C.
E-mail: tedyang@ccl.itri.org.tw

Instruction-level parallelism (ILP) consists of a family of processor and compiler design techniques that speedup execution by causing individual operations to execute in parallel. For control-intensive programs, however, there is usually insufficient ILP in a basic block for effective exploitation by ILP processors. Speculative execution is the execution of instructions before it is known whether these instructions should be executed. Thus, it can be used to alleviate the effects of conditional branches. To effectively exploit ILP, the compiler must boost instructions across branches. However, a hazard may be introduced by speculatively executed instructions that incorrectly overwrite a value when a branch is mispredicted. To eliminate such side effects, we propose a solution in this paper which uses scheduling techniques. If the result of a boosted instruction can be stored in another location until the branch is committed, the undesired side effect can be avoided. A scheduling algorithm called LESS with a renaming function is proposed for passing different identifiers along different execution paths to maintain the correctness of the program until the data is no longer used or modified. The hardware implementation for this method is relatively simple and rather efficient. Simulation results show that the speedup achieved using LESS is better than that obtained using other existing methods. For example, LESS achieves a performance improvement of 12%, on average, compared to the CRF scheme, a solution proposed recently which uses the concept of shadow register pairs.

Keywords: ILP processors, speculative execution, boosting architecture, compiler scheduling, identifier renaming

1. INTRODUCTION

Instruction-level parallelism (ILP) consists of a family of processor and compiler design techniques that speedup execution by causing individual operations to execute in parallel [25]. By the early 1990s, various ILP techniques had been adopted in advanced microprocessor design by CPU manufacturers [1, 4, 15, 22, 29]. However, since a conditional branch imposes a control dependence upon instructions that occur after the branch, the amount of parallelism that can be exploited using the present ILP techniques is limited to a sequence of instructions with exactly one entry point and exactly one exit point in the program's flow graph. Such a sequence of instructions is termed a basic block.

Received September 1, 1997; revised May 20 & September 8, 1998; accepted November 2, 1998.
Communicated by Lionel M. Ni.

For control-intensive programs, there is usually insufficient ILP in a basic block for effective exploitation by ILP processors [17]. To schedule instructions beyond the basic block boundary, instructions have to be boosted across conditional branches. *Speculative execution* is the execution of instructions before it is known whether these instructions should be executed. It is typically utilized by special hardware and/or in compiler design to alleviate the effects of conditional branches. Dynamically scheduled processors[13, 26] can use hardware branch prediction to schedule instructions from a path which is likely to be executed or to schedule instructions from both paths of a conditional branch. However, the parallelism is limited by the ability to fetch instructions, and the complicated hardware required makes this solution less attractive [2]. On the other hand, because the compiler fundamentally has the ability to optimize code schedules by analyzing critical paths from a large window of instructions and by using sophisticated instruction heuristics for scheduling [6, 9, 17, 18, 19], compiler-based speculative execution has the potential to achieve both a high instruction-per-cycle rate and a high clock rate. However, a pure compiler-based approach which can overcome the limitations of instruction scheduling does not yet exist due to the inability to manage the effects of dynamic states by means of static codes [2, 27].

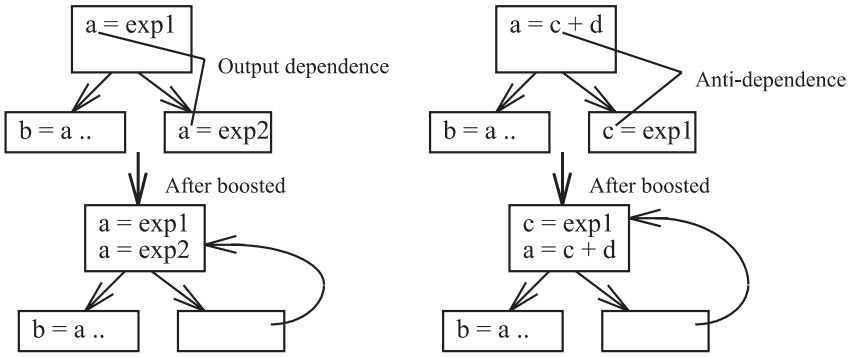
To effectively exploit ILP, a compiler must boost instructions across branches. There are two potential hazards with speculative execution [6, 27]. First, a speculatively executed instruction should not cause an exception immediately or the exception will terminate the program incorrectly. This hazard can be eliminated by suspending the handling of exceptions caused by speculative instructions until the instructions are committed [7, 21].

The other hazard is introduced by speculatively executed instructions that incorrectly overwrite a value when the branch is mispredicted. To eliminate this side effect, there are various mechanisms which allow the compiler to specify speculative execution statically. Superblock scheduling [12] of general code percolation models [6] is a pure compiler-based method which allows only instructions that will not cause side effects to be speculatively boosted across a branch. Researchers have proposed adding hardware to eliminate side effects and achieve more efficient results [2, 7, 28]. For example, Chang and Lai [7] proposed an architecture, IAS-S, with a special register file called *Conjugate Register File (CRF)* and a dedicated scheduling heuristic called *frequency-driven scheduling* to support multilevel boosting in speculative execution. However, the ability to boost is still constrained since the shadow register will force the result which is produced speculatively to be stored in a dedicated location. Moreover, although the CRF scheme has been claimed to provide a hardware solution simpler than others, the added hardware may nevertheless lengthen the instruction cycle, and the attached instructions (SetBBR, XorSR) may add further dependence on various basic blocks.

In this paper, we focus on a method for eliminating the aforementioned side effect. The main goal of this research is to find a more efficient solution which can achieve a higher degree of ILP using less hardware support than is required by the CRF scheme, the most effective solution proposed so far. We find that the side effect is caused by the object code produced by the compiler rather than by the program itself; thus, a solution can be provided using compiler scheduling techniques.

Fig. 1(a) and Fig. 1(b) show examples of undesired side effects. In Fig. 1(a), there exists an output dependence relationship between the instruction *a=exp1* and the instruction *a=exp2*. The relationship will prevent a later instruction, *a=exp2*, from being boosted

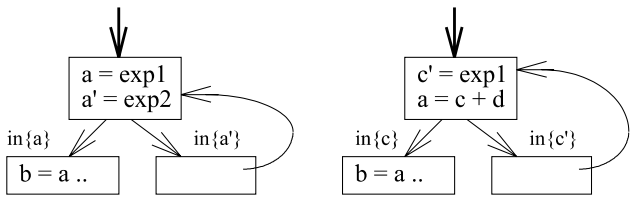
beyond the block to which it belongs. Otherwise, the side effect exhibited in the figure will cause an erroneous result in *b* if the execution flow takes the path on the left hand side. Similarly, an anti-dependence relationship, which prohibits the boosting of instructions, is shown in Fig. 1(b).



(a) Side effect caused by output dependence (b) Side effect caused by anti-dependence

Fig. 1. Examples on side effects.

Based on the examples shown in Fig. 1, it can be readily seen that the key is the destination operand of the boosted instruction. If the result of a boosted instruction can be stored in another location until the branch is committed, then the undesired side effect can be avoided. The current solutions all focus on dynamically providing a shadow location for the destination operand using dedicated hardware. These solutions assume that the operands of an instruction are fixed and can not be changed. As a matter of fact, if the compiler can rename the identifiers in order to avoid dependencies, the side effect can be eliminated without any hardware support. Fig. 2 shows a solution for the examples given in Fig.1. The identifier in the braces with the leading word, in, indicates the in set of the succeeding basic block. A compiler with a renaming function can pass on the different identifiers along the different execution paths to maintain the correctness of the program until the data is no longer used or modified.

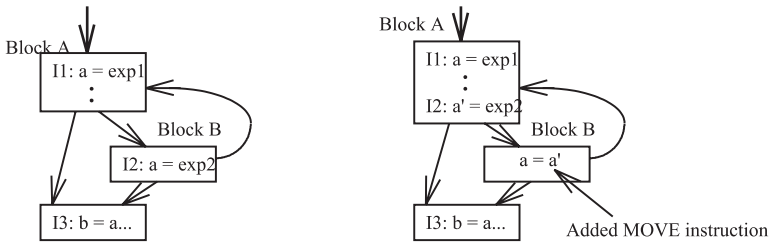


(a) Renaming for output dependence (b) Renaming for anti-dependence

Fig. 2. Side effects eliminated by renaming.

Although the concept of identifier renaming is a simple idea and has been studied extensively in many systems [14], there are still challenges which must be overcome before it can be realized to support speculative execution. First, traditional renaming is only one part of the front end of the compilation transformation used to eliminate anti/output dependence without considering its interaction with the hardware environment, e.g., the size of the register file. It may actually cause the performance of a program to deteriorate by increasing the amount of spill code if the function can not cooperate well with the later register allocation process.

Second, the traditional renaming algorithm proposed previously is mainly dependent on the data flow analysis and, thus, tied to the *Use-Definition*(UD Set) and *Definition-Use* (DU Set) chains. This algorithm is not suitable for speculative execution since the instructions in the chains are all assigned to the same identifier. For example, the variables in instructions *I*1, *I*2 and *I*3 in Fig. 3(a) can not be renamed in order to benefit from speculative execution because *I*3 belongs to DU(*I*1, *a*) and DU(*I*2, *a*) at the same time. If the instruction *I*2 is to be boosted to the basic block *A* with a different identifier, then the compiler will not be able to find the correct identifier for *I*3 since there are two different execution paths to *I*3. However, there are other solutions which can be adopted to provide a solution to this problem. For example, by inserting a MOVE instruction into basic block *B* where the instruction *I*2 belongs, as shown in Fig. 3(b), the semantics can be correctly retained. To probe further, a new solution is proposed in this paper to eliminate the added MOVE instructions using special designated hardware to dynamically fetch the correct data according to the execution flow.



(a) Three instructions in a flow graph

(b) Renaming for speculative execution by inserting a MOVE instruction

Fig. 3. An example of identifier renaming.

Since the identifier renaming scheme depends primarily on the techniques of compiler scheduling, related issues are discussed in section 2. A hardware enhancement to support identifier renaming is presented in section 3, and a scheduling algorithm called *LESS* is proposed in section 4 for implementation of identifier renaming. The algorithm is based on the idea of frequency-driven scheduling in cooperation with pre-pass and post-pass scheduling. Section 5 presents preliminary simulation results obtained using the execution scheme and a comparison with the CRF scheme. In addition to the impressive performance improvement shown by the simulation, some interesting phenomena revealed by the results are also discussed in that section. Finally, a conclusion and directions for future work are presented in section 6.

2. IDENTIFIER RENAMING AND RELATED ISSUES IN ILP SCHEDULING

Code scheduling is a technique that arranges the instruction sequence of a program so as to minimize the number of execution cycles. In previous works [9][18], trace scheduling was proposed to allow code motion beyond the basic blocks. After all the intermediate optimizations have been performed, the edges of the flow graph of the operations are attached to the expected execution frequencies generated by profiling. The modified flow graph is divided into several execution paths, called traces. Each trace is an acyclic sequence of basic blocks in the control flow graph. Traces are selected and scheduled according to the frequencies of execution. A variant of trace scheduling, termed superblock scheduling, was later proposed in [12]. A superblock is a trace that has no side entrance. Control may enter only from the top but may leave from one or more exit points.

Although trace/superblock scheduling can schedule a linear sequence of basic blocks that is most likely to be executed, it has been criticized because it focuses on the current trace and neglects the rest of the program. More specifically, there are two defects in the method of trace/superblock scheduling. First, although the code is scheduled along the most frequent execution path, the instructions which are scheduled across several blocks are probably not the most suitable instructions for such scheduling. This is because the combined probabilities of the instructions to be executed decrease according to the distance between blocks. For example, an instruction in block *D* shown in Fig. 4 may be boosted to block *A* with a probability of 0.36 since there are two branches on the execution path with a probability of 0.6 each. This probability of 0.36 is lower than the probability for block *C*, the block directly following block *A*. However, the instructions in block *C* can not be boosted into block *A* using the trace concept.

The other defect in trace/superblock scheduling is introduced by the concept of boundary scheduling. In order to simplify the complexity of scheduling, the trace/superblock compiler divides a sequence of blocks, along with the most frequently executed path, into a trace as the largest basic block for scheduling. The tail of the block will lose it's chance to speculatively execute the following instructions that belong to other blocks and are immediately behind, but do not belong to, the trace. Let us use Fig. 4 as an example. In this

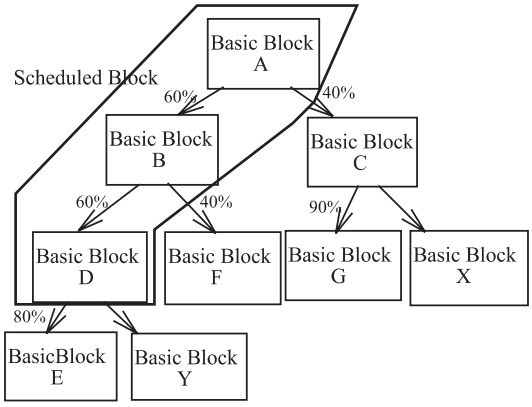


Fig. 4. An example of trace scheduling in a control flow graph.

figure, blocks *A*, *B*, and *D* are grouped as a trace. The instructions in blocks *B* and *D* may be promoted and boosted into block *A*. However, the instructions in block *D* that can not be boosted will spend cycles for execution and will not be able to promote other instructions in block *E*, which will probably be executed after block *D*, to fill in available cycles.

Recently, a scheduling strategy was proposed to schedule the most likely executed instructions from both paths of a branch [19, 30]. Furthermore, Chang and Lai [7] implemented this strategy and proposed an algorithm, called *frequency-driven scheduling*, which can greedily schedule basic blocks by boosting the instructions from the succeeding blocks of all execution paths without the constraints on block boundaries. The algorithm schedules basic blocks according to the precedence order of the control flow graph. A block, say *A*, can be selected for scheduling when all of its preceding blocks have been scheduled. The scheduler will promote the instructions that can be executed speculatively from the blocks succeeding block *A* in order to fill the instruction slots in the block. When block *A* is scheduled into ILP form and the contents of the succeeding blocks are modified because of speculative scheduling, the scheduler can then select another block for scheduling until all the blocks have been scheduled. Although the scheduler will spend more time than the traditional trace/superblock scheduler, the algorithm can efficiently eliminate the aforementioned defects and produce a more compact ILP code. For these reasons, this algorithm was selected as the framework for our study on scheduling which incorporates identifier renaming and a mapping scheme.

We will now address the following identifier renaming related issues with respect to speculative execution: How should program semantics be maintained? At which stage during compilation should renaming be performed? Are there any drawbacks introduced by the technique of register renaming? How can the drawbacks introduced by renaming be conquered? Does the scheme of identifier renaming indeed provide a better solution than the others?

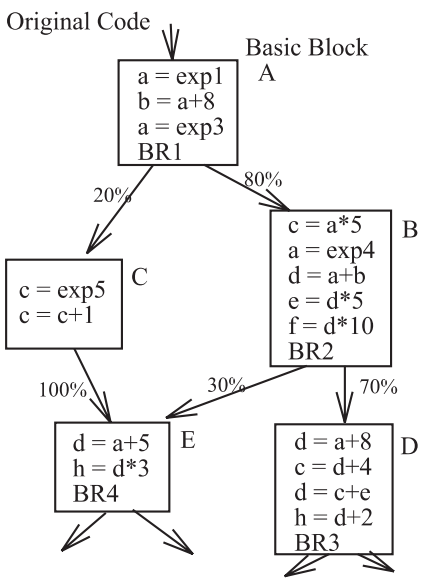
When an identifier is renamed, the new identifier must be transmitted to the following basic blocks. The in sets and the out sets of the corresponding basic blocks must be modified to transmit the renaming relationships by inserting/deleting elements into/from the corresponding in sets and out sets of each basic block. Once the instructions in a basic block have been scheduled in ILP form, the operands can be modified in order to maintain the correctness of the program semantics by checking the in sets definition of the block and the out sets definition of the preceding blocks. The relationships obtained through renaming can then be transmitted to the in sets of the following blocks until the identifier is no longer used or redefined.

Renaming is accomplished by adding extra identifiers which replace the original identifiers, so this must be done before the identifiers are allocated to dedicated locations. This means that renaming must be completed before register allocation. Moreover, identifier renaming is performed in order to eliminate the side effects produced by speculative scheduling. This function is implemented in the code scheduling phase. This implies that any scheduling involving the function of renaming must also be performed before register allocation.

Speculative scheduling reorders the sequence of instructions and causes the range of alive identifiers to be lengthened. Furthermore, renaming increases the use of identifiers, which will incur additional dependence relationships as well. It may also increase the demand for registers since a larger number of identifiers will be alive at the same time, which in turn may result in register spill.

To alleviate these drawbacks, compilers must be able to use a limited number of registers more efficiently. A heuristic solution to this problem, based on the idea of two-pass scheduling, was proposed in [7]. In contrast to the conventional two-pass scheduling approach, in which both scheduling passes actually modify the instruction sequence, our solution performs pre-pass scheduling only in constructing of the conflict graph of the ideal ILP schedule for the later register allocation phase. When the registers are all used up, the register allocator can ignore the conflict edges introduced by speculative scheduling and allocate some distinct identifiers to the same register without inserting a spill code. Thus, during post-pass scheduling, the effect of speculative scheduling due to generation of these identifiers will not be allowed to execute speculatively. This method can be modified to eliminate the effect of identifier renaming under the constraint of limited registers. The details of this approach will be presented in section 4.

The identifier renaming scheme does provide better solutions, as demonstrated by this example. Fig. 5 shows a portion of a sequential program. Assuming that the number of operation slots in each instruction cycle is four, and that the latency of each instruction is 1, the final parallel form of the scheduled code produced by the local scheduler is presented in the figure. From the scheduled sequence, we can expect that it will take an average of 7.92 cycles to complete code execution.



ILP Code

```
Basic Block A : 2 Cycles
   a = exp1
   b = a + 8      a = exp3      BR1
Basic Block B : 3 Cycles
   c = a * 5      a = exp4
   d = a + b
   e = d * 5      f = d * 10    BR2
Basic Block C : 2 Cycles
   c = exp5
   c = c + 1
Basic Block D : 4 Cycles
   d = c + 8
   c = d + 4
   d = c + e
   h = d + 2      BR3
Basic Block E : 2 Cycles
   d = a + 5
   h = d * 3      BR4
```

***Execution Cycles:**

Path A->B->D: 2+3+4 = 9 **Path A->B->E: 2+3+2 = 7**

Path A->C->E: 2+2+2 = 6 **Average Cycles:9*0.56+7*0.24+6*0.2 = 7.92**

Fig. 5. A portion of the sequential code and its ILP counter part after local scheduling.

Fig. 6 shows the parallel form, as scheduled using code motion with renaming, corresponding to the example shown in Fig. 5. The superscripts <′> and <″> indicate that renaming has taken place. The letter in parenthesis after an the operation stands for the original block to which the operation belonged. We can find from the codes that the average number of cycles has been reduced from 7.92 to 5.04. This is 1.57 times faster than local scheduling. The instruction added in block <$B \rightarrow E$> is a MOVE operation used to revise the renamed identifier so that the following instructions will maintain correct semantics.

In Fig. 6, we can find that the preceding blocks, *A* and *B*, can be efficiently filled with speculative operations by means of identifier renaming. If other operations exist behind the succeeding blocks, *C*, *D* and *E*, then the final schedule can be expected to be even more efficient than Fig. 6 shows.

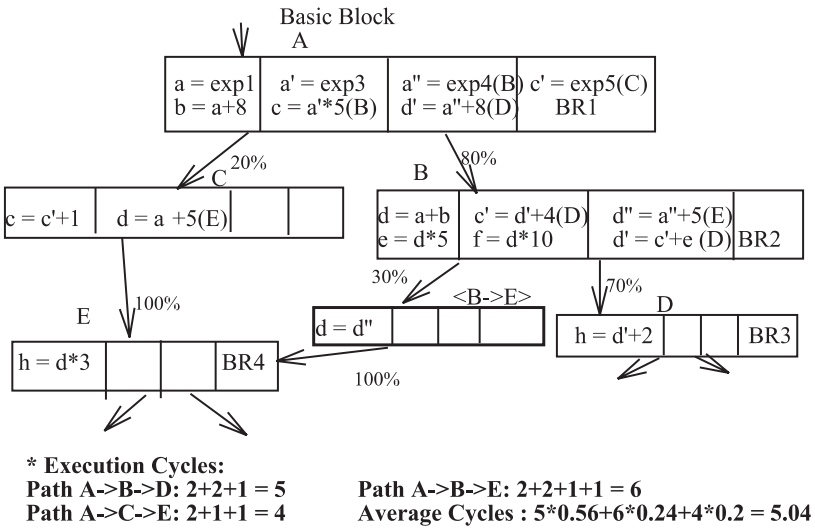


Fig. 6. ILP by applying renaming to the example shown in Fig. 5.

To investigate the renaming scheme further, we selected the CRF scheme [7] for comparison. Fig. 7 shows the final parallel code scheduled using that scheme. Since the CRF scheme is based on the idea of the shadow register with the use of register pairs, the final code is exhibited in the form of registers. Because of the inability to boost due to the dedicated shadow register and the added instructions, SetBBR/XorSR, for adjusting the CRF scheme, a total of 6.4 cycles is needed for execution. The speedup is 1.24, which is nowhere near the speedup provided by the previous identifier renaming scheme. It should be pointed out that the approach using shadow register pairs, as implemented in the CRF scheme, allows only one instruction to be boosted at a time for instructions having the same destination operand. As shown in the example, the technique of identifier renaming, when properly applied, can efficiently eliminate anti and output dependencies, alleviate the constraints introduced by the original program semantics and lead to a scheduling result which is more efficient than other solutions.

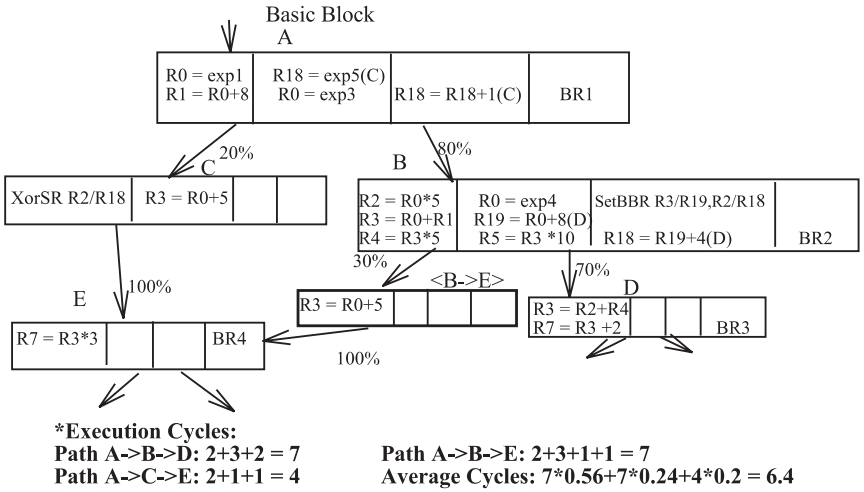


Fig. 7. ILP by applying CRF scheme from the example in Fig. 5.

3. ARCHITECTURAL SUPPORT FOR RENAMING

Let us look at Fig. 6 again. Since the instruction, $h = d * 3$ in block E, uses a source operand d which has been allocated to different identifiers according to the execution paths, the compiler needs to add a MOVE instruction to maintain the semantics. An instruction of this type, that is, an instruction which provides a source operand whose value may be variant to execution paths, is called an intertwined instruction. If the number of intertwined instructions is large, the inserted MOVE instructions may add extra overhead required for execution and degrade the parallelism.

In order to improve the imperfection caused by intertwined instructions, a hardware-based mapping scheme is proposed to dynamically fetch the correct source data without the addition of MOVE instructions. Assume that there are N registers in the processor. By building a mapping register with a width of lgN bits, we can set the mapping register with different values corresponding to the different execution paths to record the correct path that has actually been executed. An intertwined instruction can fetch the correct data by mapping the source operand along with the mapping register to generate the correct register number. Because the map is built using a line of XOR gates as shown in Fig. 8, it is much simpler than other hardware solutions.

In the mapping scheme, the general instructions can fetch their operands by means of the source register number directly without mapping. The compiler can identify the intertwined instruction and assign a special register number for mapping. Because the different values of the source are generated from different execution paths, the mapping register can be assigned with different values using a SetMR instruction to assign a value to the mapping register before the block is committed, and the correct data can be fetched by XORing the logical number and the value of the mapping register. As an example, Fig. 9 shows the scheduled code modified from Fig. 6 by using the mapping scheme. Assume that the operand d referenced by the intertwined instruction, $h = d * 3$, is assigned to registers *R3* and

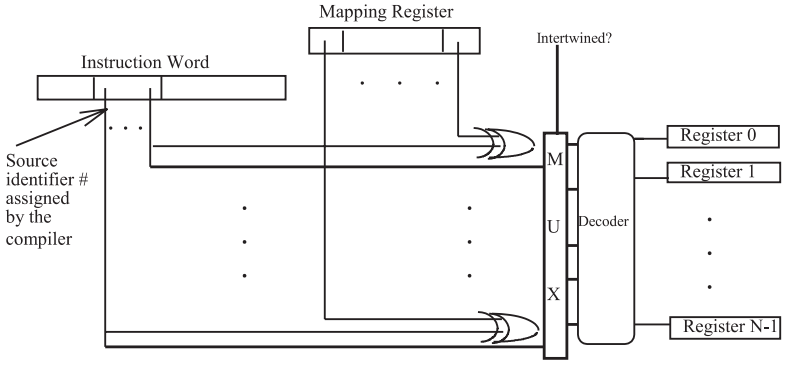


Fig. 8. Mapping scheme for fetching correct register data.

R20 before block *E* is committed. The compiler can assign the values 00000_2 and 10111_2 to the mapping register corresponding to the execution paths $C \rightarrow E$ and $B \rightarrow E$, respectively. By XORing the mapping register and the logical number when the intertwined instruction is executed, the correct data will be fetched dynamically from register *R3*(00011 = 00000 XOR 00011) or *R20*(10100 = 10111 XOR 00011) with the correct data.

From the scheduled codes shown in Fig. 9, we find that the average number of execution cycles is further reduced to 4.8, and that an impressive speedup of 1.65 is achieved. It should be noted that although there is still an added instruction, SetMR, the total amount of added instructions has been effectively reduced. Regardless of the amount of intertwined data, because the intertwined instructions in the same basic block can be assigned the same mapping number, the added SetMR instruction set for defining the value of the mapping register is limited to, at most, one for each basic block and is not proportional to the amount of intertwined data that a pure compiler-based solution used to insert MOVE instructions for each instance of such data. Moreover, the SetMR instruction can be scheduled in any slot of the basic block without a dependence constraint as the original solution.

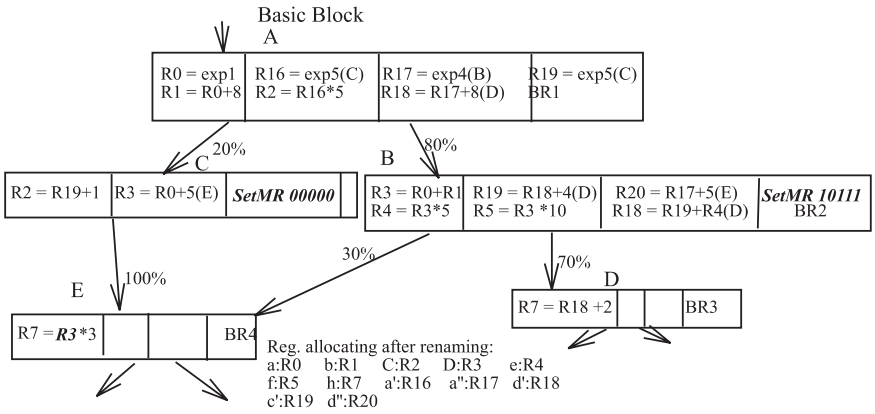


Fig. 9. ILP by applying register renaming along with the mapping scheme to the example in Fig. 5.

Examining the mapping scheme more closely, we find that although the scheme can dynamically fetch the correct data using the logical register number of the intertwined instruction and the value of the mapping register, this same data may be used later in the following basic blocks with new values in the mapping register. If the same data is continuously fetched using this scheme, the compiler will have to maintain the correct semantics by setting different logical register numbers for the instructions in different blocks. The overhead could be too much for the compiler if the situation of intertwined instructions occurred many times and the data was used in many blocks following the original block that contained the intertwined instruction. To eliminate this overhead, we further propose an assistant mechanism that can automatically move the intertwined data into a dedicated register during execution of the first intertwined instruction.

If the intertwined data is the first operand of the instruction, then when the intertwined instruction is executed, the data will be moved into a register whose binary representation is identical to the binary representation of the destination register of the instruction but different in the most significant bit. For example, if the destination is *R7* (00111) in a register file with 32 registers, the register that latches onto the intertwined data is *R23* (10111). On the other hand, if the intertwined data is the second operand of the instruction, the data will be moved into a register whose binary representation is identical to the binary representation of the destination register of that instruction but different in the second most significant bit. For the above example with *R7* (00111), the register that latches onto the intertwined data is *R15* (01111). To simplify the description of this scheme, we will refer to these registers as d+16/d+8 in the following paragraphs. By incorporating the automatic move into the design of this mapping scheme, the compiler can assign intertwined instructions with fixed register numbers to match different values without the overhead of an added MOVE instruction. In this manner, the compiler can also use the intertwined data accurately in the succeeding instructions.

The requirement for the inclusion of information about whether an operand is intertwined calls for a change in the instruction format and, thus, changes the *instruction set architecture* (ISA). A similar requirement exists in most of the researches on speculative execution. For example, the predicated execution scheme requires that information be embedded in the instruction in order to indicate the predicated register number. The shadow register scheme needs information to specify the destination of the shadow register where the result should be stored. Although further research in this area is needed, two possible solutions are outlined in the following paragraphs.

1. The mapping scheme can be further modified so as to assign each register with a mapping field in the mapping register. By replacing the global mapping register with the fields, the needed information can be adequately and correctly provided to the compiler. In this way, the automatic move is no longer needed. Furthermore, this approach eliminates the need for information to indicate whether the data is intertwined because all the operands of instructions are treated as intertwined data and are mapped using the mapping scheme.
2. By expanding the instruction set to include some intertwined instructions for the provision of intertwined information, the conventional instructions can be viewed as normal instructions without being intertwined, and the requirements for ISA changes are eliminated. This solution is somewhat similar to the application of partial predicated execution [20] to solve the same problem in the predicated execution.

Using this mapping scheme and the previously described move mechanism, the compiler can manage various situations efficiently in order to achieve unconstrained scheduling.

4. LESS SCHEDULING

Although our algorithm is based on the *frequency-driven scheduling* steps described above, the kernel of our algorithm is very different from that used in *frequency-driven scheduling*. The provision of identifier renaming is the primary reason for the differences. For example, identifiers may be split into several items when they are renamed for boosting, in which case the dependence relationships between identifiers will be changed at the same time. Furthermore, the mapping scheme will force some identifiers to be assigned with values according to predetermined allocation relations for registers. The changes introduced by renaming and the constraints forced by the mapping scheme must be incorporated into our algorithm. Since the algorithm proposed in this paper schedules basic blocks using three step-by-step functions, it is called lift-each-step-scheduling, or LESS.

The main challenge of LESS is to rename identifiers so as to eliminate false data dependencies and reduce the amount of added compensation code for handling intertwined instructions. An efficient register allocation is of primary concern. As discussed in section II, the renaming function must be performed before register allocation. As a consequence, the renaming procedure (or function) may produce extra data dependencies due to the insufficient number of registers. This insufficiency will cause extra spill codes to be inserted into the original codes and, thus, may lead to poor performance. A two-pass scheduling technique, as shown in Fig. 10, has been designed. The GNU C compiler(GCC) is used as the front end of the compiler, and a set of classic optimizations, such as common subexpression elimination, constant propagation, loop optimization, and peephole optimization are performed in this phase. The GCC output is a sequential code with infinite symbolic identifiers. The first step in LESS is to send the codes to a register allocator to acquire profiling information.

In the second step, the pre-pass scheduler is active to schedule the codes in ILP form with infinite registers using the renaming techniques. In the pre-pass step, the scheduler will produce an alias graph that records the relationships among identifiers. The alias graph is a weighted directed graph that contains two types of edges. The first type of edge is called a white edge and is represented as $<a, b>$ with a weight x. This means that the identifier a is an alias for the original symbol b for speculative execution, and that the weighted value x is the probability that the data, a, produced by the speculative instruction is correct in the execution. Furthermore, any intertwined data is also recognized in pre-pass scheduling, and a second type of edge, called a black edge, is added. A black edge $<d+16/d+8, d>$ identifies the relationship between the intertwined data and destination data of an intertwined instruction with a weighted value of 16/8 to indicate that the register number should be $d+16/d+8$ for the data d.

Using the alias graph and the live-range-conflict graph produced by the scheduled codes, the register allocator can then allocate a register to each identifier by means of the graph-coloring heuristic algorithm at the beginning of step 3. When there are not enough registers to be assigned for all identifiers that may be living, the allocator can eliminate the edges and recombine some nodes in the conflict graph according to the white edges with

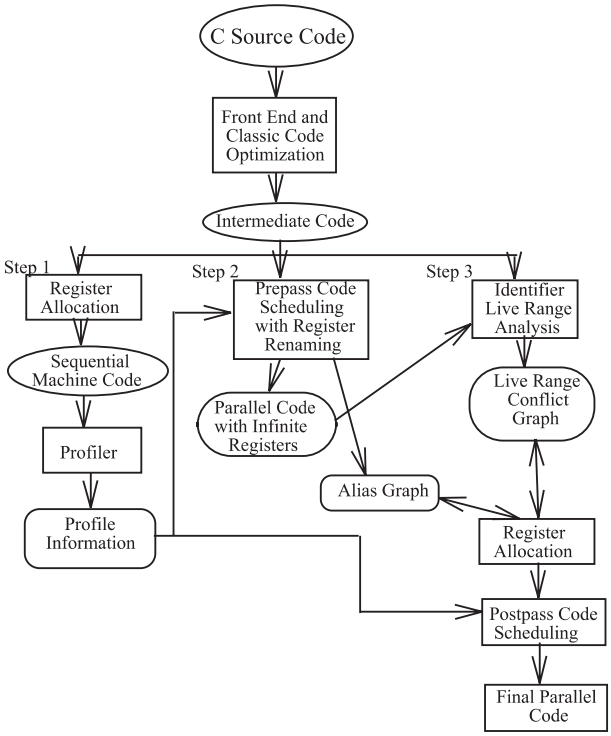


Fig. 10. Two-pass scheduling in LESS, Lift-Each-Step-Scheduling.

less probability weight exists in the alias graph to reduce the demand for registers and to avoid spill code insertion. However, speculative codes from the most probable execution paths are still maintained. The black edges that record the constraints of register numbers are checked when the register allocator assigns registers to identifiers. If there are not enough register pairs for the vertices of a black edge, the compiler can insert a MOVE instruction in order to replace the function of the intertwined instruction. After all of the identifiers are allocated to dedicated registers, a post-pass ILP scheduler is activated in order to produce the final ILP code. As a final requirement, SetMR instructions are inserted into the blocks in order to maintain the functions of the mapping scheme.

The nitty-gritty of LESS is pre-pass scheduling. As described above, LESS schedules sequential codes in the same manner as *frequency-driven scheduling* does. That is, each block is scheduled according to the precedence of the control flow graph. The three major functions in the LESS algorithm are as follows.

1. Transformation(Basic block *A*):
   Modify the identifier symbols that have been renamed in the preceding blocks and recognize the intertwined instructions in block *A*.
2. Promotion(Basic block *A*, Instruction *I*):
   Schedule the instruction *I* in block *A* as early as possible by means of renaming.
3. Boosting(Basic block *A*, Basic block *B*):
   Boost the instructions in block *B* to block *A* according to the execution path by means of renaming.

A brief description of the main functions of LESS is presented below. Algorithm 1 is the main procedure for LESS, and Algorithms 2, 3, and 4 represents the three functions of Transformation, Promotion and Boosting, respectively.

Algorithm 1: Pre-pass LESS Algorithm
Step 1: Select a basic block *A*, whose predecessors are all scheduled. If not found then goto Step 6.
Step 2.1: Use the Transformation function to modify the identifiers in block *A* by means of the in set of the block and the out sets of the predecessors.
Step 2.2: Use the Transformation function to find the intertwined instructions in block *A* and add black edges into the alias graph.
Step 3: List-schedule the instructions in block *A* using the Promotion function to rename identifiers and add white edges to the alias graph with weight 1.
Step 4: While (there are idle operation slots in block *A*)
Step 4.1: Select a basic block *B* that is a successor of block *A* where the probability of execution path from *A* to *B* is greater than a threshold. If not found, then goto Step 5.
Step 4.2: Boost instructions in block *B* using the Boosting function to fill up the idle slots in *A*.
Step 4.3: While (there is a boosted instruction that has been renamed)
Step 4.3.1: An element, say $x' \rightarrow x$, will be inserted into the out sets of blocks in the execution path $A \rightarrow B$ in order to record the renaming, and an element is also inserted into the in set of block *B*. A white edge is also added into the alias graph by setting the weight with the probability of the execution path $A \rightarrow B$.
Step 5: Goto Step 1.
Step 6: End Algorithm

Algorithm 2: Transformation (Basic block *A*)
Step 1: Select an alias pair $s^* \rightarrow s$ from the in set of block *A* and delete the record; if not found then goto Step 5.
Step 2: If (there exists only one definition for the identifier *s* in the out sets of the predecessors of the block)
Step 2.1: If (*s* is not defined in block *A*)
Step 2.1.1: Transmit the alias relation to the succeeding blocks by means of the in/out sets of the related blocks.
Step 2.2: Replace the identifier s of the instructions in block *A* with *s** until the instruction which redefines *s* is reached.
Step 3: Else
Step 3.1: Find the first instruction *I* which uses *s* as a source operand; if not found then insert a MOVE instruction to move *s* to identifier *d*.
Step 3.2: Mark *I* as an intertwined instruction.
Step 3.3: Replace the identifier *s* of the instructions behind instruction *I* with *d*+16/*d*+8 until the instruction which redefines *s* is reached.
Step 3.4: Add the alias pair $d+16/d+8 \rightarrow s$ into the in/out sets of the related blocks.
Step 3.5: Add the black edges <*s*, *d*+16/*d*+8> and <*d*+16/*d*+8, *d*> with weights 1 and 16/8, respectively, into the alias graph.

Step 4: Goto Step 1.
Step 5: End

Algorithm 3: Promotion (Basic block *A*, Instruction *I*)
Step 1: Find the instruction which is the predecessor of *I*, and which has an anti/output dependence with *I*, and call it I_j. If not found then goto Step 9.
Step 2: Find the destination of I_j ,*d*, that has been used in instruction *I* and rename the identifier *d* of *I* as *d**.
Step 3: If(*d* is not redefined in the block *A* and *d* exists in the out set of block *A*)
Step 3.1: Replace *d* with $d^* \rightarrow d$ in the out set of block *A*.
Step 3.2: Add $d^* \rightarrow d$ to the in sets of all the successors of block *A*.
Step 4: Replace the identifier *d* of the instructions that are the successors of instruction *I* in block *A* with *d** until the instruction which redefines *d* is reached.
Step 5: Modify the data precedence graph.
Step 6: Add the white edge <*d**, *d*> with weight 1 into the alias graph.
Step 7: If(instruction *I* is an intertwined instruction and there exists a black edge<*d*+16/*d*+8, *d*> in alias graph)
Step 7.1: Replace the black edge with <*d*+16/*d*+8, *d**>.
Step 8: Goto Step 1.
Step 9: End

Algorithm 4: Boosting (Basic block *A*, Basic block *B*)
Step 1: Insert a null basic block, say B', between the block **B** and the predecessors of *B* but not in the execution path of $A \rightarrow B$.
Step 2: If (there exists an idle operation slot in block *A*)
Step 2.1: Select an instruction, *I*, such that no predecessors of *I* exist in the block which can be scheduled into the slot of *A*. If not found, then goto Step 4.
Step 2.2: If (there exists a source operand *s* which is defined in the execution path $A \rightarrow B$)
Step 2.2.1: Abandon the instruction and goto Step 2.1.
Step 2.3: Schedule instruction *I* into block *A* and insert the instruction into the blocks B'.
Step 2.4: If(there exists an anti/output dependence between instruction *I* and some instruction, say I_j, which belongs to a block in the execution path)
Step 2.4.1: Rename *d* of instruction *I* as *d**.
Step 2.4.2: Add *d** into the in/out sets of the basic blocks in the execution path.
Step 2.4.3: Add $d^* \rightarrow d$ into the in set of block *B*.
Step 2.4.4: Replace identifier d of the instructions in block *B* with *d** and add $d^* \rightarrow d$ to the in sets of all the successors of the block.
Step 2.4.5: Add the edge <*d**, *d*> with a weight which is the probability of execution from *A* to *B* into the alias graph.
Step 3: Goto Step 2.
Step 4: If (the added block $B' == \emptyset$), then delete the block.
Step 5: End.

5. PRELIMINARY SIMULATION RESULTS

Seven benchmark programs were used to evaluate the performance improvement gained through identifier renaming and to verify the correctness of the LESS algorithm. As a comparison, the same benchmarks were also scheduled using the CRF algorithm. The benchmarks were quicksort(qs), the ackerman function(ack) and pattern matching proposed by Knuth, Morris and Pratt(kmp), the 8 queen problem(8q), the word count utility(wc), the algorithm of all pairs shortest paths(short), and the sieve benchmark. The features of these benchmarks can be classified into three groups as shown in Table 1.

Table 1. The category of benchmarks.

Benchmarks	Feature
ack, qs	Small recursive program
wc, short	Nested loops
sieve, 8q, kmp	Complex conditional execution in nested loops

The architectural models used for the simulations were built based on DLX[24]. When a benchmark was compiled into the DLX assembly code using GNU C, we first inserted, by hand, increment instructions and added instructions onto the beginning of each basic block to measure the execution frequencies of the blocks. The code was rescheduled using LESS and CRF under the assumption of hardware parallelism with 2, 4, and 8 issues. The code was then simulated using the superscalar version of the DLX simulator, superdlx, to obtain execution time statistics. The hardware environment was assumed to have one delay slot for each load and each branch instruction, and to contain a 20-entry load buffer and a 10-entry store buffer. The latencies of the integer/floating point operations followed the assumptions about the DLX architecture.

In the progress of code scheduling, some existing imperfections in the original design of the LESS algorithm were found and modified. They included the method for scheduling the code in loops and a method for bookkeeping the speculative code when the basic block that the boosted instruction belonged to had two or more preceding basic blocks. In the original version, the algorithm was designed to schedule a loop just like a basic block, and to attempt to boost the operations of the first iteration out of the loop. The scheduler must insert a basic block to keep track of the boosted code to ensure correctness. However, the gain from the solution was not obvious, and the regularity of the loop could be destroyed, leading to poor performance than other solutions, such as software pipelining [16]. Finally, the algorithm was modified to view a loop as a closed basic block, that is, a code portion to be scheduled without speculative execution. Furthermore, the bookkeeping for the boosted instructions was also modified. In the original design, when an instruction was boosted from a basic block, say *A*, to a preceding basic block, the instruction also had to be duplicated onto the paths from other preceding blocks to *A*. Duplication was completed by creating a new block that contained the boosted instruction. This method always caused a branch instruction to be added to the end of the new basic block so as to direct the execution

flow to *A*. This action lengthened the execution path. We provided a simple solution to this problem. When an instruction was boosted, a new basic block was not created, but the boosted instruction was duplicated in the other preceding basic blocks, regardless of whether the block had been scheduled in the ILP form or not. If the preceding block had been scheduled, the instruction was added onto the end of the block. Although this could increase the execution of the preceding block by one cycle, the penalty was reduced since there was no added branch. Furthermore, if the preceding block was not scheduled, the compiler could still be able to schedule the added instruction without increasing the number of execution cycles.

The benchmarks were all scheduled using the modified LESS algorithm and the CRF scheme under the assumption of three issue-rates. Table 2 lists the number of instructions in each case renamed by the modified LESS algorithm and the CRF scheme. The numbers clearly show that LESS can eliminate more unnecessary data dependencies than CRF.

Table 2. Numbers of instructions renamed.

Benchmarks	qs	ack	kmp	8q	wc	short	sieve
Renaming by LESS	7	4	19	43	23	11	22
Renaming by CRF	5	1	7	23	14	7	21

Speculative execution is likely to demand a greater number of registers. The requirement is expected to be more in LESS since there are no physical shadow registers and the boosting distance is always comparatively longer than that in other schemes. If the requirement exceeds what the hardware can support, spill code must be inserted to allocate data into memory, leading to poorer performance. The maximum numbers of registers used concurrently for the benchmarks by LESS are shown in Table 3. The table indicates that the required number of registers is indeed increased. However, the result shows that the working set did not increase dramatically, and that a set of 32 registers is still adequate.

Table 3. Amount of registers needed.

Benchmarks	qs	ack	kmp	8q	wc	short	sieve
Registers used in original codes	10	8	9	8	15	16	17
Registers used in LESS codes	13	11	21	28	28	23	25

Table 4 shows the various speedups for the codes executed with no speculative execution, with CRF scheduling and with LESS scheduling, over the base architecture for issue-rates of 2, 4, and 8. The base architecture used for comparison is a pipelined DLX with an issue-rate of 1. It is noted that since the load/store buffers are eliminated in the base architecture, the speedups can be higher than the issue-rate since the ILP architectures contain the buffers.

Table 4. Speedups over base architecture.

Issue-Rates.	2							4							8						
benchmarks	qs	ack	kmp	8q	wc	short	sieve	qs	ack	kmp	8q	wc	short	sieve	qs	ack	kmp	8q	wc	short	sieve
Original Speedup	3.0	3.6	2.8	2.7	2.1	2.7	2.7	3.3	4.9	2.9	3.2	2.1	2.9	2.9	3.5	5.3	3.0	3.4	2.1	2.9	3.2
CRF Speedup	3.1	3.9	2.8	3.3	2.7	3.1	3.0	3.4	5.3	3.8	4.4	3.0	3.1	3.3	3.6	5.7	3.8	4.6	3.0	3.1	3.3
LESS Speedup	3.1	4.1	2.8	3.6	2.8	3.2	3.1	3.4	5.3	4.6	4.6	3.0	3.4	3.7	3.6	5.7	5.6	5.6	3.0	3.4	3.7

Based on Table 4, we can make the following observations.

1. Speculative execution can improve the performance of a program with potential parallelism among its basic blocks.

The kmp and 8q benchmarks are two examples with obvious parallelism potential that can be exploited using speculative execution because of their complex control structure. The performance improvement obtained using speculative execution can be over 66% and 83% when the issue-rate is 8. However, when the hardware parallelism is not so high, say, when the issue-rate is only 2, the performance improvement is degraded as shown in kmp. Since the parallelism that exists in the basic blocks of kmp is already high enough to occupy the operation slots, there is little room left for speculative scheduling. However, 8q shows that when the parallelism in a basic block is not enough to fill the slots, speculative execution can still perform well.

The performance of benchmarks usually reflects their behavior. The factors in the qs and the ack benchmarks that limit the effect of speculative execution are:

A. They are recursive algorithms that contain a large number of load/store operations in the small kernel for passing parameters and saving/reloading local data. In contrast, there are only a few loads/stores that will prevent speculative scheduling in the codes for kmp and 8q.

B. There are loops in the codes that can not be scheduled for speculative execution. On the other hand, although there are some loops in the kmp and 8q benchmarks, the basic blocks in the loops can still be scheduled for speculative execution.

2. For the selected benchmarks, LESS always outperforms CRF, and the improvement is more evident when speculative execution can significantly benefit the program's execution.

We see in Fig. 11 that, when a benchmark is suitable for speculative execution, LESS always outperforms the CRF scheme and achieves an average improvement of 12% for the following reasons. The ability to boost instructions in the LESS algorithm is not constrained by the naming of data produced by the compiler. However, this is not true for the CRF scheme due to the inability of register pairs. The kmp benchmark with a performance improvement of 48%, and the 8q benchmark, with an improvement of 21%, are two such examples.

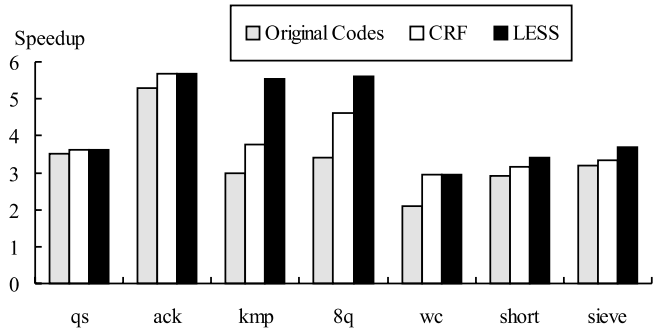


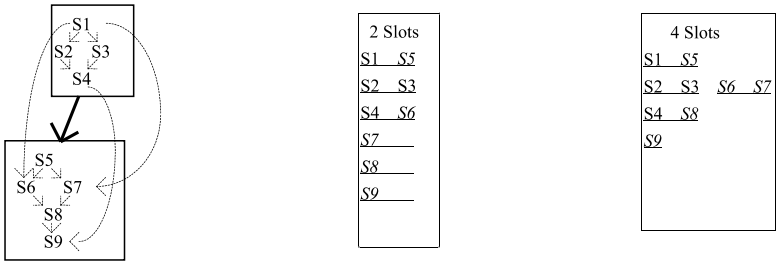
Fig. 11. Speedups of the three parallel models with an issue-rate of 8.

The added instructions in the CRF scheme lengthen the execution. The situation becomes crucial for the insertion of XorSR instructions since the instruction must be added to the beginning of the committed basic block and the related instruction is, thus, delayed. For the sieve program, the added instructions cause a degradation of 10% in the speedup ratio.

In the remaining portion of this section, the impacts of intertwined instruction and the issue-rate are discussed based on the simulation results.

The effect of intertwined instruction is not evident in the execution under scheduling by LESS. There are only two intertwined instructions in the 8q benchmark and five intertwined instructions in the sieve benchmark. However, in the execution of these two programs, the added instructions did not effect the execution time because they were inserted into adequate null slots.

There are various versions of each of the benchmarks which are scheduled for simulation under various issue-rates. However, some simulation results show that the scheduled codes introduced to obtain higher issue-rates may perform more efficiently than codes scheduled for lower issue-rates even though the hardware simulation environment is built with a lower issue-rate. For example, the codes of 8q scheduled for issue-rate 8 take 122644 cycles under an environment with an issue-rate of 4, but the same codes scheduled for an issue-rate of 4 need 131529 cycles for execution in the same environment. This anomaly appeared in the simulations for the models of both the CRF scheme and LESS scheduling. By assuming that the latency of each instruction is 1, and that the hardware environment is set up with an issue-rate of 2, the anomaly can be explained as shown in Fig.12.



(a) Dependence graph of a code portion (b) Parallel codes scheduled for anissue-rate of 2 (c) Perallel codes scheduled for anissue-rate of 4

Fig. 12. A code portion scheduled for 2 issue and 4 issue.

Fig. 12.(a) shows a portion of the code with two successive basic blocks. The data dependence relationship among the instructions is exhibited by the dotted lines in the figure. If the code is scheduled for an issue-rate of 2, as shown in Fig. 12.(b), the execution takes 6 cycles on hardware with an issue-rate of 2. However, if the code is scheduled for an issue-rate of 4, as shown in Fig. 12.(c), the execution takes 5 cycles on hardware with an issue-rate of 2. This means that by merging two or more basic blocks and by scheduling the codes as they are in the same basic block, a more efficient portion of codes can be produced since the critical paths of the blocks can be overlapped.

6. CONCLUDING REMARKS

We have proposed in this paper a new method for speculative execution that can exploit the ILP existing in a program efficiently using the technique of LESS compiler scheduling. LESS is based on the technique of identifier renaming. By combining this algorithm with a mapping register and an automatic move mechanism for intertwined instruction, the proposed method can achieve higher performance than schemes that perform speculative execution by means of shadow registers. Preliminary simulation results show that the speedups achieved using LESS always outperform those introduced by register pairs. LESS can be used with available ILP processors without changing the hardware environment to achieve a higher degree of parallelism.

Several issues are worth further investigation. The inclusion of software pipelining/ superblock scheduling to benefit the execution of loops and sequential codes is one such issue. In another direction, by examining the ILP codes scheduled using LESS, we find that the codes of the basic blocks in later portions of a program are usually boosted into earlier basic blocks. This means that many basic blocks with branch instructions are only clustered in the later portions of the program. If hardware for concurrent branches, that is, multiway branch architecture [4], can be added to the speculative execution scheme and integrated into a dedicated scheduling technique, such as branch merging [8], the resulting performance can be expected to be further improved. In yet another direction, because boosting of Load/Store instructions is limited due to memory ambiguation, the memory disambiguation ability [10, 11, 23] is also important to improve the performance of speculative execution. If LESS can be in order modified to classify some dedicated memory references models, such as the references of array elements and the passing of parameters, speedup can be further improved using speculative execution.

REFERENCES

1. D. Alpert and D. Avnon, “Architecture of the pentium microprocessor,” *IEEE Micro*, Vol. 13, No. 3, 1993, pp. 11-21.
2. H. Ando, C. Nakanishi, T. Hara and M. Nakaya, “Unconstrained speculative execution with predicated state buffering,” in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, 1995, pp. 138-149.
3. S. Arya, H. Sachs and S. Duvvuru, “An architecture for high instruction level parallelism,” in *Proceedings of the 28th Annual Hawaii International Conference on System Sciences*,

1995, pp.153-162.
4. G. R. Beck and D. W. L. Yen, "The cydra 5 minisupercomputer: architecture and implementation," *Journal of Supercomputing*, Vol. 7, No. 1/2, 1993, pp. 143-180.
5. P. P. Chang et al., "The importance of prepass code scheduling for superscalar and superpipelined processors," *IEEE Transactions on Computers*, Vol. 44, No. 3, 1995, pp. 353-370.
6. P. P. Chang, N. J. Warter, S. A. Mahlke, W. Y. Chen and W. W. Hwu, "Three architectural models for compiler-controlled speculative execution," *IEEE Transactions on Computers*, Vol. 44, No. 4, 1995, pp.481-494.
7. M. C. Chang and F. P. Lai, "Efficient exploitation of instruction-level parallelism for superscalar processors by the conjugate register file scheme," *IEEE Transactions on Computers*, Vol. 45, No. 3, 1996, pp. 278-293.
8. C. M. Chen, C. T. King and Y. Y. Chen, "Branch merging for scheduling concurrent executions of branch operations," *IEE Proceedings–Computers and Digital Techniques*, Vol. 143, No. 6, 1996, pp. 369-375.
9. J. A. Fisher, "Trace scheduling: a technique for global microcode compaction," *IEEE Transactions on Computers*, Vol. C-30, No. 7, 1981, pp. 478-490.
10. D. M. Gallagher, W. Y. Chen, S. A. Mahlke, J. C. Gyllenhal and W. W. Hwu, "Dynamic memory disambiguation using the memory conflict buffer," in *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating System*, 1994, pp. 183-193.
11. A. S. Huang, G. Slavenburg and J. P. Shen, "Speculative disambiguation: a compilation technique for dynamic memory disambiguation," in *Proceedings of the 21st Annual International Symposium on Computer Architecture*, 1994, pp. 200-210.
12. W. W. Hwu et al., "The superblock: an effective technique for VLIW and superscalar compilation," *Journal of Supercomputing*, Vol. 7. No.1/2, 1993, pp. 229-248.
13. Q. Jacobson, S. Bennett, N. Sharma and J. Smith, "Control flow speculation in multiscalar processors," in *Proceedings of the 3rd Annual International Symposium on High-Performance Computer Architecture*, 1997, pp. 218-229.
14. V. E. Kotov, "Automatic construction of parallel programs," *Algorithms, Software and Hardware for Parallel Computers*, 1984.
15. A. Kumar, "The HP PA-8000 RISC CPU," *IEEE Micro*, Vol. 17, No. 2, 1997, pp. 27-32.
16. M. S. Lam, "Software pipelining: an effective scheduling technique for VLIW machines" in *Proceedings ACM SIGPLAN'88 Conference on Programming Language Design and Implementation*, 1988, pp. 318-328.
17. M. S. Lam and R. P. Wilson, "Limits of control flow on parallelism," in *Proceedings of the 19th Annual International Symposium on Computer Architecture*, 1992, pp. 46-57.
18. P. G. Lowney, S. M. Freudenberger, T. J. Karzes, W. D. Lichtenstein, R. P. Nix, J. S. O'Donnell and J. C. Ruttenberg, "The multiflow trace scheduling compiler," *Journal of Supercomputing*, Vol. 7, No. 1/2, 1993, pp. 51-142.
19. S. A. Mahlke, D. C. Lin, W. Y. Chen, W. W. Hwu, B. R. Rau and R. A. Bringmann, "Effective compiler support for predicated execution using the hyperblock," in *Proceedings of the 25th Annual International Symposium on Microarchitecture*, 1992, pp. 45-54.
20. S. A. Mahlke, R. E. Hank, J. E. McCormick, D. I. August and W. W. Hwu, "A compari-

son of full and partial predicated execution support for ILP processors," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, 1995, pp. 138-149.
21. S. A. Mahlke, W. Y. Chen, W. W. Hwu, B. R. Rau and M. S. Schlansker, "Sentinel scheduling for VLIW and superscalar processors," in *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating System*, 1992, pp. 238-247.
22. E. McLellan, "The alpha AXP architecture and 21064 processor," *IEEE Micro*, Vol. 13, No. 3, 1993, pp. 36-47.
23. A. Moshovos, S. E. Breach, T. N. Vijaykumar and G. S. Sohi, "Dynamic speculation and synchronization of data dependences," in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, 1997, pp. 181-193.
24. D. A. Patterson and J. L. Hennessy, *Computer Architecture: A Quantitative Approach*, Second edition, San Mateo, California: Morgan Kaufmann, 1996.
25. B. R. Rau and J. A. Fisher, "Instruction-level parallel processing: history, overview, and perspective," *Journal of Supercomputing*, Vol. 7, No. 1/2, 1993, pp. 9-50.
26. J. E. Smith, "Dynamic instruction scheduling and the astronautics ZS-1," *IEEE COMPUTER*, Vol. 22, No. 7, 1989, pp. 21-35.
27. M. D. Smith, M. S. Lam and M. Horowitz, "Boosting beyond static scheduling in a superscalar processor," in *Proceedings of the 17th Annual International Symposium on Computer Architecture*, 1990, pp. 344-255.
28. M. D. Smith, M. Horowitz and M. S. Lam, "Efficient superscalar performance through boosting," in *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating System*, 1992, pp. 248-259.
29. S. P. Song et al., "The PowerPC 604 RISC microprocessor," *IEEE Micro*, Vol. 14, No. 5, 1994, pp. 8-17.
30. A. K. Uht, "Extraction of massive instruction level parallelism," *Computer Architecture News*, Vol. 21, No. 3, 1993, pp. 5-12.

Lei Wang (王壘) is a Ph.D. candidate in the department of Information Engineering at Feng Chia University. He is also a lecturer in the department of Electrical Engineering at FCU. His interests include compilers, instruction level parallelism, architectures, and fault tolerance.

Ted Chun-Chung Yang (楊濬中) received his B. S. degree in Electrical Engineering from National Cheng-Kung University and Ph.D. degree in Computer Science from the Illinois Institute of Technology in Chicago. He engaged in research and development at Lockheed Research Laboratory, Palo Alto, California; Bell Laboratories, Naperville, Illinois; and Motorola, Inc., Schaumburg, Illinois. He joined the faculty of Feng Chia University in 1982 and served as FCU president from 1988 to 1995. He is currently General Director of the Computer and Communications Research Laboratories/ITRI in Hsinchu. His research interests include computer architecture, fault-tolerant systems, logic and system simulations, and communication systems.