Enhancing Multiple-Path Speculative Execution with Predicate Window Shifting *

Jenn-Yuan Tsai
Department of Computer Science
University of Illinois
Urbana, IL 61801 USA
j-tsai1@uiuc.edu

Pen-Chung Yew
Department of Computer Science
University of Minnesota
Minneapolis, MN 55455 USA
yew@cs.umn.edu

Abstract

Speculative execution has long been used as an approach to exploit instruction level parallelism across basic block boundaries. Most existing speculative execution techniques only support speculating along *single* control path, and heavily rely on branch prediction to choose the right control path. In this paper, we propose an extended predicated execution mechanism, called predicate shifting, to support speculating along *multiple* control paths. The predicate shifting mechanism maintains a condition/predicate window for each basic block. With the condition/predicate window, instructions can be guarded by predicates related to current or future branch conditions. The predicate shifting mechanism can reduce the number of required tag bits by shifting conditions/predicates out of the condition/predicate window whenever they are no longer in use. To incorporate the predicate shifting mechanism into a VLIW processor, a new result-buffering structure, call future buffer, is used to buffer uncommitted results and to evaluate predicates. The FIFO structure of the future buffer not only simplifies exception handling but also allows multiple uncommitted writes to the same register. Experimental results show that the predicate shifting mechanism can use predicate tag effectively and achieve 24% performance improvement over the previous predicating mechanism [2] using a small predicate tag.

Keywords: instruction level parallelism, speculative execution, predicated execution, VLIW processor architecture.

*This work is supported in part by the National Science Foundation under Grant No. MIP 93-07910, MIP 94-96320, and CDA 9502979; by the U.S. Army Intelligence Center and Fort Huachuca under Contract DABT63-95-C-0127 and ARPA order No. D 346; and by a gift from Intel Corporation.

1 Introduction

Superscalar and VLIW architectures have become dominant in high performance processor design. They both provide multiple instruction decoders and functional units to exploit instruction level parallelism (ILP). Previous studies [21, 8] have shown that the ILP within a basic block is very limited. Thus, exploiting ILP across basic block boundaries is essential to achieve higher performance. Speculative execution, which allows the execution of instructions before their dependent branch conditions are resolved, is a widely used approach for eliminating control dependences and allowing instructions to be moved across basic block boundaries. Instruction movement and scheduling for speculative execution can be done at compile time via global scheduling, or at run time via branch prediction and dynamic scheduling, or both.

Without any hardware support, a compiler can only perform limited speculative movement in order to maintain safety and legality of the program execution. To support speculative execution, some *result-buffering* mechanism for uncommitted results is usually needed. Reordering buffers [7, 16] in superscalar processors and shadow register files [2, 19, 18] in VLIW processor provide such result-buffering mechanism for speculative execution.

Besides result-buffering, most existing speculative execution techniques also adopt *branch prediction* [9]. In superscalar architectures, branch history table [7] is often used to predict branch results dynamically. In VLIW architectures, compilers use profile information [5, 3] to identify the most-frequently executed trace. With branch prediction, compilers or processors can speculatively move or execute instructions along the most-frequently taken control path. Limiting speculative execution to a single control path has the benefit of maximizing the execution efficiency from the most-frequent executed path as well as simplifying the hardware for instruction fetching and result buffering. For programs with low branch prediction accuracy, however, the performance may suffer significantly because the compilers or processors cannot exploit much parallelism from the unpredicted paths which are actually taken at run time. To improve performance for processors with a high issue rate on programs with a low prediction accuracy, one may consider using multiple-path speculative execution to exploit ILP from all control paths.

Predicated execution [11, 12, 15] or *guarded execution* [6, 13] has been used as a way to eliminate branches from instruction streams. In predicated execution, a conditional branch is converted into a predicate defining instruction. Instructions which are control dependent on the conditional branch are then guarded by the predicate. By eliminating branch instructions, predicated execution can effectively increase basic block size and reduce branch delay cycles. However, using predicated execution alone still does not allow instructions which are control dependent on a conditional branch to be moved before their predicate defining instruction. In other words, the control dependences are merely converted to data dependences rather than avoided as in the case of speculative execution.

Speculative execution and predicated execution can be used together to support multiple-path speculative execution [2, 20]. Predicated execution provides a good mechanism to represent control dependences in different control paths. With the support of result buffering mechanisms, instructions can be speculatively moved and executed before their predicate defining instructions. Recently, Ando et al. [2] proposed a predicated state buffering mechanism called *predicating* which can support multiple-path speculative execution. They show that *predicating* can gain more speedup than single-path speculative execution. The mechanism, however, requires a unique condition name for each conditional branch in a region[1]. Each instruction also requires a predicate tag to specify control dependences on all of the branch conditions in the region. The size of the predicate tag thus limits the number of conditional branches allowed in a region. As a result, this constraint can limit the size of a region and its exploitable ILP.

In this paper, we propose a more flexible mechanism, called *predicate shifting*, which can support both multiple-path speculative execution and predicated execution without the limitations mentioned above. The predicate shifting mechanism uses a predicate window to address the predicate currently in use. It can keep the predicate tag size small by shifting predicates out of the predicate window whenever they are no longer in use. In addition, we present a new result-buffering structure, called *future buffer*, for VLIW architectures. The future buffer can evaluate the predicate states of uncommitted results according to the run time conditions. Differing from the shadow register files used in [2, 19, 18], the future buffer adopts a first-in, first-out (FIFO) structure to simplify the handling of speculative exceptions and allow multiple uncommitted writes to the same register.

In the rest of this paper, we review some previous works in section 2. In section 3, we describe our scheme and its required architectural and compiler support. Section 4 shows some performance results. Finally, in Section 5, we present our conclusions.

2 Background

Without hardware support for speculative execution, the compiler must take full responsibility to ensure the correctness of speculative instruction movement. It must be very conservative and, hence, can only perform limited speculative movements which are *safe* and *legal*. A speculative instruction movement must be safe to avoid speculative exceptions that may alter the result of the program execution. A speculative instruction movement must also be legal to avoid writing to a register or a memory location whose value may be used by instructions on other control paths [18].

[1] A region is a set of basic blocks that includes a entry basic block, which dominates all other basic blocks in the region[1].

These restrictions can severely limit the exploitable ILP.

Hardware mechanisms to buffer uncommitted results and to handle speculative exceptions can relax such restrictions. To allow speculative instruction movement along multiple control paths, additional hardware is needed to handle future conditions and control dependences. In the following subsections, we review existing architectural approaches for these mechanisms.

2.1 Buffering Uncommitted Results

Some speculative instruction movements may become illegal because the new values generated by those speculative instructions may overwrite the values that are still live on other control paths. This can be avoided by renaming the destination registers of those speculative instructions. This method, however, requires additional instructions to copy the new values to the original registers if the control paths containing the renamed registers are taken. The performance of such register renaming is limited by the availability of free registers.

Smith et al. [19, 18] propose *boosting* which uses a shadow register file and a shadow store buffer to store the results of the speculative instructions until the branch instructions they are control dependent on become committed. If the branches are predicted correctly, the processor updates the machine state with the results in the shadow structures, otherwise, the results in the shadow structures are discarded. To support n levels of branch speculation, the processor must provide n shadow register files. This mechanism allows unconstrained speculative execution in a *single* control path.

The *predicating* mechanism [2] uses a similar shadow structure to buffer speculative results from multiple control paths. Each entry of the register file contains a predicate field and two data fileds, one for sequential (committed) value and the other one for speculative value. A flag W is used to indicate which field is storing the current sequential value. When the processor executes a speculative instruction, the predicate tag of the instruction is copied to the predicate field of the destination register, and the execution result is stored in the speculative data field. If the predicate evaluates to be true in later execution, the W flag is flipped and the speculative value becomes the current sequential value. This shadow register structure requires only one shadow register file for any level of branch speculation. However, it allows at most one uncommitted speculative write to any register, because there is only one shadow register for each register.

2.2 Handling Speculative Exceptions

Because of the speculative instruction movement, some exceptions which will not occur in the original execution may occur during the speculative execution. The simplest hardware mechanism to handle such speculative exceptions is to convert all speculative instructions which could cause exceptions into non-excepting or silent version of those instructions. When an exception occurs to a silent instruction, instead of signaling the exception, the instruction write a polluted result to its destination register. The continuing execution of the program may lead some later non-speculative instruction to read the polluted value. The non-speculative instruction will signal the exception when it reads the polluted value. Colwell et al [4] use a *NaN* (i.e. Not-A-Number in the IEEE floating point standard) to represent the polluted result of a speculative instruction which causes exception. The use of a *NaN* by a non-speculative instruction will signal the exception. There are several shortcomings with this method. First, it is not guaranteed to signal an exception if the polluted result is conditionally used. Also, it is very difficult to locate and re-execute the original instruction which causes the exception.

Mahlke et al. proposed a *sentinel* scheduling model [10] to accurately detect and report all exceptions caused by speculative instructions. Sentinel scheduling model divides each speculative instruction into two parts, the non-excepting part that performs the actual speculative operation, and the sentinel part that signals an exception, if necessary. The sentinel part always remains in the original basic block of the speculative instruction. If the execution of the non-excepting part causes an exception, the processor writes the excepting address into the destination register and sets the exception tag of the register. Any later speculative instructions that use the register will copy the excepting address and the exception tag to their destination register. The detection of the exception will be postponed until the execution of a non-speculative instruction which uses the register with the exception tag set, or until the execution of the non-speculative sentinel part of the speculative instruction which causes the exception. With the excepting address stored in the source register, the processor can accurately locate the speculative instruction which causes the exception.

Boosting [19, 18] and predicating [2] also postpone speculative exceptions until the commit point. Instead of saving the excepting address in the destination register, they mark the exception flag in the corresponding shadow register when an exception occurs. Later, when the processor is committing the shadow value to the sequential register, the exception is detected and signaled. When the exception is signaled, the processor discards all speculative data in shadow registers, and starts the exception handling process. After handling the exception, the processor must restart the execution of the program. In boosting, the exception handler uses the address of the committing branch to index a jump table and then jump to the recovery code associated with the committing branch. The jump table and the recovery code (which contains the instructions to be re-executed at each commit point) are both generated by the compiler. In predicating, the processor re-executes all speculative instructions which are yet to be committed. It uses a special execution mode called *recovery mode* to differentiate recovery execution from normal execution.

2.3 Representing Future Condition and Control Dependencies

To support multiple-path speculative execution, we must provide a way to represent future branch conditions and the control dependences related to the future branch conditions that a speculative instruction is dependent on. By encoding the control dependences in instructions, the compiler can inform the hardware in what conditions to commit or to squash the results of the instructions. The representation of future conditions and control dependences must be easy to manipulate and to encode.

Boosting [19, 18] supports speculative execution along a single control path, that is, only instructions in the most-frequently taken direction of each conditional branch can be speculatively moved above the branch instruction. Since only one control path is possible at any time, the control dependences of a speculative instruction can be encoded as a count of the number of the conditional branches it has been moved above. The representation of control dependences in boosting is very simple and efficient, but it also restricts the opportunities of exploiting ILP from other control paths.

Predicating [2] supports speculative execution along multiple control paths. In this scheduling model, instructions from both directions of a conditional branch can be moved above the branch instruction. Predicating mechanism gives each conditional branch in a region a unique condition name. The predicate tag of an instruction specifies its control dependences related to all the branch conditions in the region. The execution result of an instruction can be committed if and only if its predicate evaluates to be true with respect to all the branch conditions in the region. By predicating an instruction with all branch conditions in the region, the hardware can easily decide whether the instruction should be committed or squashed. The drawback of this mechanism is that the number of conditional branches in a region is limited by the size of the predicate tag. Since the speculative execution doesn't exploit ILP across region boundaries, limiting the size of a region will reduce the exploitable ILP in the region.

3 The Predicate Shifting Mechanism

In this section, a multiple-path speculative execution mechanism, called *predicate shifting*, is described. Like predicating, predicate shifting allows a compiler to perform unconstrained speculative code motion along multiple control paths. This mechanism, however, uses a different scheme to represent control dependencies and their corresponding predicates of a speculative instruction. The basic idea is, instead of using the boundary of a region, we use a *condition/predicate window* to specify branch conditions and their corresponding predicates currently in use. When a branch condition and its corresponding predicate are no longer in use, they will be shifted out of the condition/predicate window, and the succeeding future branch condition and its predicate are shifted in. In a sense, the condition/predicate window is being moved downward. The size of the predicate tag that represents a window only limits the size of the window. It will not limit the number of conditional branches in a region. Additionally, the proposed scheme provides a new mechanism to buffer speculative results. This result-buffering mechanism can simplify the handling of speculative exceptions, and allows multiple uncommitted results to be written into the same register. In the following subsections, the execution model, the hardware support, the exception handling mechanism, and the compiler support are described.

3.1 Execution Model

In our model, global instruction scheduling is performed within a *region*. A region is a set of basic blocks that includes an entry basic block which dominates all other basic blocks in the region [1]. Similar to [2], tail duplication is performed to make every basic block in a region, except the entry block, having only one predecessor in the control flow graph. To simplify global scheduling and tail duplication, our region formation algorithm will not include loops or function calls inside a region. After tail duplication, every conditional branch inside a region is converted into a *condition defining* instruction (called SETC) followed by a predicated jump to the branch target. The value of a condition can be *(1) true (branch taken)*, *(2) false (branch not taken)*, or *(3) undefined (unknown yet)*. The control dependency related to a condition is represented by a predicate, which can be *(1) execute only if the condition is true (branch taken)*, *(2) execute only if the condition is false (branch not taken)*, or *(3) don't care.* The execution or committing of an instruction which is control dependent on the branch condition is guarded by its predicates. An instruction can be guarded by several predicates as long as the predicate tag is wide enough to hold them. For an instruction guarded by multiple predicates, the execution result of the instruction can be committed if and only if all its predicates evaluate to true.

In our execution model, there is no unique name (address) or absolute storage location associated with each branch condition and its corresponding predicate. Instead, we use a *condition/predicate window.* The condition/predicate window represents the scope of contiguous conditions and their predicates with respect to each basic block. The number of entries in a condition/predicate window is equal to the number of predicates in the predicate tag. For each basic block, the condition window logically contains conditions[2] defined in the basic block and in basic blocks on the succeeding control

[2]Since our model supports predicated execution, several basic blocks can be fully predicated into a larger basic

```
     ld    r6, r29(30)
     alu   r3, r0, imm
     ld    r2, r6(12)
     alu   r5, r29, imm
     alu   r15, r2, imm
     alu   r24, r2, imm
     beq   r15, r0, LC

     alu   r25, r2, imm
     bgt   r24, r0, LA

     alu   r1, r0, imm
     alu   r8, r2, imm
     beq   r25, r1, LA

     alu   r9, r8, imm
     st    r6(12), r9
     jmp   LC
LA:  beq   r3, r0, LB

     alu   r4, r6, r0
     ret

LB:  alu   r4, r0, imm
     ret

LC:  ld    r4, r29(52)
     st    r29(32), r3
     st    r29(48), r6
     ret
```

Figure 1: A code segment from *compress*.

paths. The first entry in the condition window is the condition defined by the first SETC instruction in the basic block[3]. The second entry in the condition window is the conditions which immediatedly follow the first condition in the control flow graph, and so on. Instructions in a basic block thus can use the predicate window to specify their control dependencies in the window.

Figure 1 is a code segment from Spec92 benchmark *compress*. The corresponding code after tail duplication and branch conversion is shown in Figure 2. In the restructured code, all conditional branches have been converted into SETC/JMP pairs. The condition window for each basic block is shown on the upper-left corner of each basic block. Condition *Cn* in a condition window denotes the *nth* level branch condition counted from the first basic block L1. For basic block L2, the first entry of its condition window is the condition defined by i10 (denoted by C2). However, the second entry is the condition defined by either i14 or i28 (denoted by C3). They can be denoted by the same condition C3 because the execution of these two instructions are always exclusive. After the execution of basic block L2, the condition C2 is shifted out of the window. Either i14 in L3 or i28

block, which may contains several conditions.

[3]After instruction scheduling, the first SETC instruction may be moved up to the last instruction cycle of the predecessor basic block.

Pn	Predicate
0 1	Execute only if Cn = False
1 0	Execute only if Cn = True
1 1	Don't care

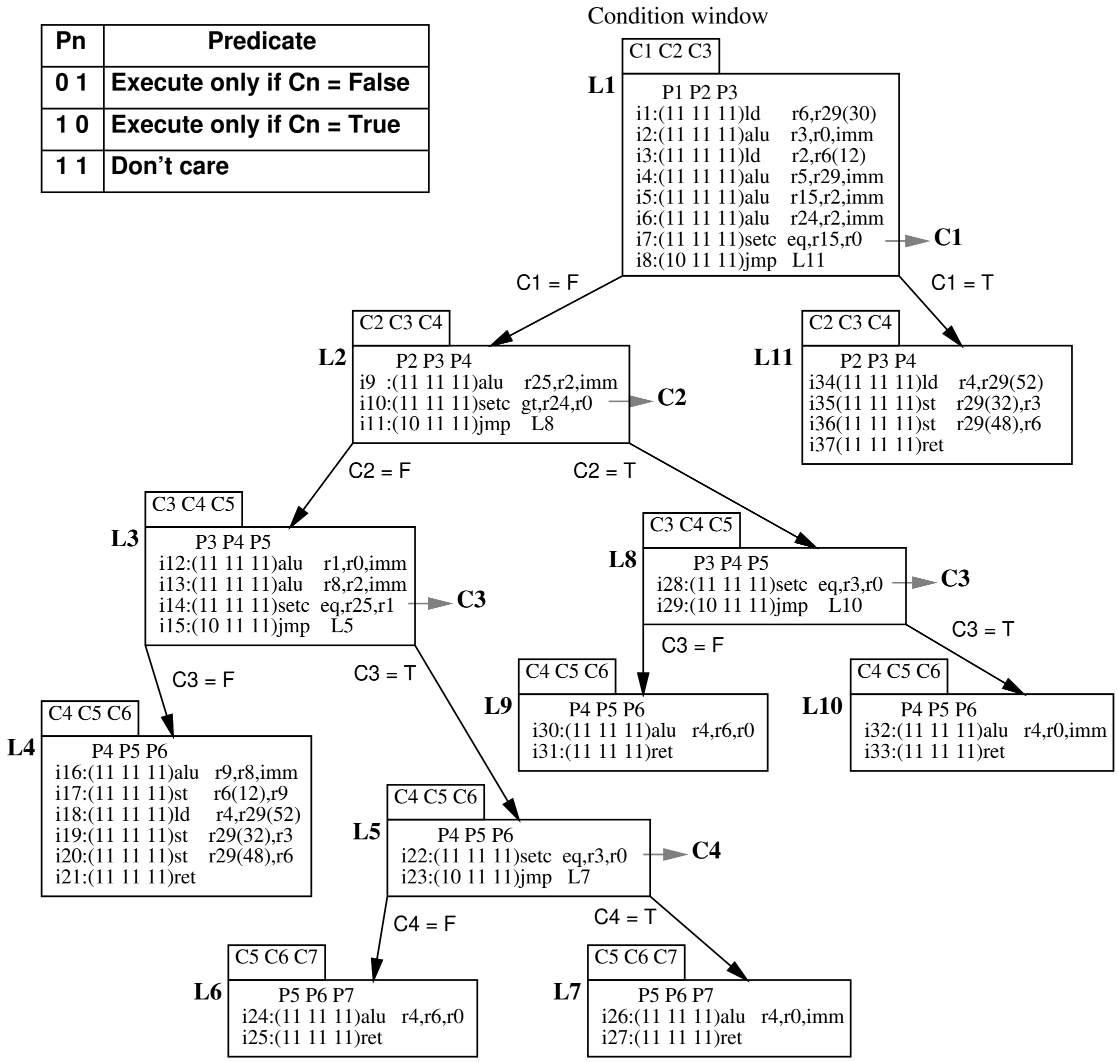


Figure 2: The code segment from *compress* after tail duplication and branch conversion.

in L8 will be executed and define C3, which becomes the first entry of the condition window.

Figure 2 also shows the predicate tag associated with each instruction. Here, we assume the hardware can support 3 predicate levels. Each predicate requires two tag bits to represent the following control dependency relations.

- 0 1 : Execute only if the corresponding condition is false.
- 1 0 : Execute only if the corresponding condition is true.
- 1 1 : Don't care.
- 0 0 : Squashed (used by hardware only)

The code in Figure 2 is before global scheduling. There is no speculative instruction movement yet. All instructions except the predicated jumps are independent of any *current conditions* and *future conditions*, hence, all have a predicate tag of (11 11 11). A condition is said to be current with respect to an instruction if its value is known and still in use when the instruction is issued. A condition is said to be a future condition if its value is not yet defined when the instruction is issued. To move an instruction above a SETC instruction, the compiler must update its predicate tag to indicate its control dependencies on the condition defined by the SETC instruction. For example, if i30 in basic block L9 is moved to L1, its predicate tag will become (01 10 01), which means the committing of i30 is dependent on C1(false), C2(true), and C3(false).

At run time, the processor also maintains a dynamic condition window to hold current conditions and future conditions. The dynamic condition window is the same as the condition window viewed at compile time. The condition window is updated when the processor crosses a basic block boundary. The processor identifies a basic block boundary when it encounters a branch instruction[4]. When the processor decodes a branch instruction, the conditions defined in the current basic block are retired and shifted out of the condition window, and new future conditions are shifted in (see Figure 2). Note that a condition defined by a SETC instruction issued as the last instruction of the current basic block is regarded as the condition defined in the next basic block, so it will not be shifted out of the condition window when the current basic block is terminated.

Since the condition window can contain both current conditions and future conditions, the compiler can encode the control dependencies of an instruction on future conditions (for speculative execution) as well as on current conditions (for predicated execution). In this mechanism, unlike predicating [2], the predicate tag only contains control dependencies on conditions in current use instead of on all conditions in a region. The width of predicate tag only limits the levels of branch predication and speculation instead of the size of a region. This allows instruction scheduler to form regions of larger sizes for more ILP.

3.2 Architectural Support

To efficiently support the model described in the subsection 3.1, we need to provide mechanisms for buffering execution results, and committing or squashing the results according to the run-time conditions.

Figure 3 shows the block diagram of a generic four-issue processor. Like other VLIW processors, this processor provides multiple decoders and functional units for multiple-instruction issue and

[4]In our execution model, all basic blocks are terminated by a branch instruction (including function call and return) after full tail duplication

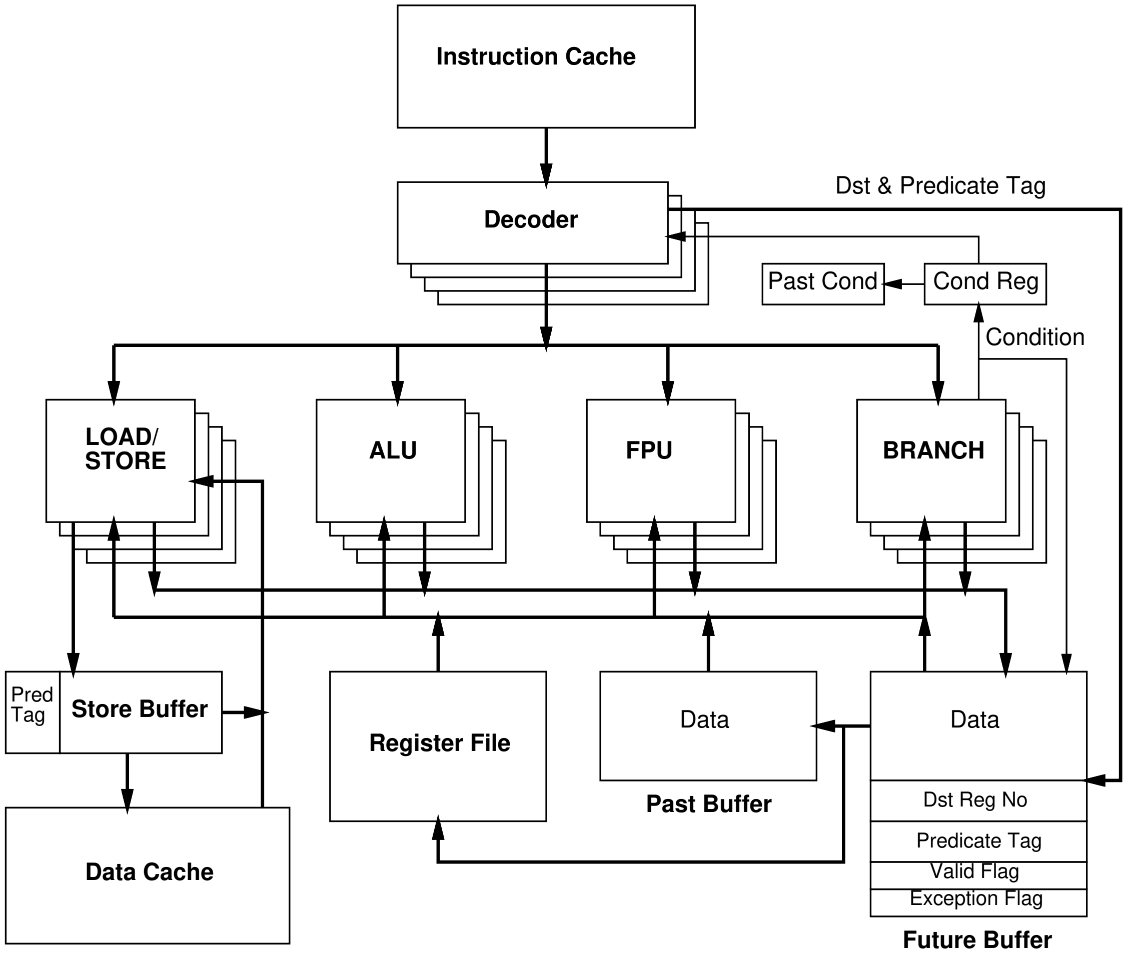
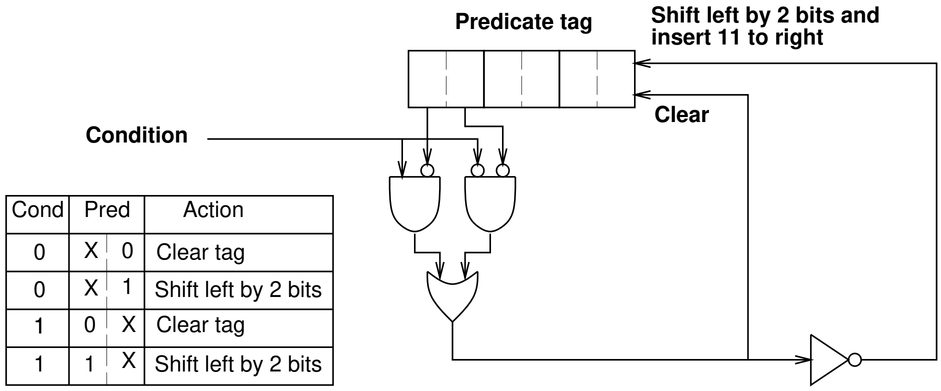


Figure 3: The processor block diagram

execution. The *condition register* above the branch units is a shifting register which stores the values of conditions in the current condition window. The *past-condition register* beside the condition register is also a shifting register. It stores the values of recently retired conditions from the condition register. The retired condition values are needed for exception recovery.

This processor model provides a *future buffer* to buffer uncommitted results. The future buffer is organized as a first-in, first-out (FIFO) queue to allow instructions to be completed and committed in order. By committing instructions in the program order, the processor can greatly simplify the exception handling and recovery. The mechanism for exception handling is described in the next section. The future buffer contains a data field for storing execution results, a register-name field for specifying the destination registers of the execution results, a predicate tag field for storing the predicate state associated with each result. In addition, each entry in the future buffer includes



Cond	Pred		Action
0	X	0	Clear tag
0	X	1	Shift left by 2 bits
1	0	X	Clear tag
1	1	X	Shift left by 2 bits

Figure 4: Actions taken on predicate fields of a future buffer entry when executing SETC instruction

a valid flag to indicate whether the result is ready or not, and an exception flag for signaling the exception caused by the instruction associated with the entry. Like the past-condition register, there is a *past buffer* for saving the recently committed data being shifted out of the future buffer. The past buffer is also used for exception recovery. The number of entries in the past buffer must be equal to the number of entries in the future buffer.

When the processor starts executing a program, all entries in the condition register are initialized to *undefined*. At this point, all conditions in the condition window are future conditions. When a decoder fetches and issues instructions, the decoder allocates an entry at the top of the future buffer for each instruction and copies its destination register name and the predicate tag into the corresponding fields in the entry. After the execution, the result of the instruction is written into the data field of the allocated entry, and the valid flag is set to true. The predicate state of an instruction is initially assigned by the compiler as described in the previous subsection. During execution, the predicate state will be evaluated and updated by the processor according to the condition values defined by the SETC instruction. Figure 4 shows the hardware mechanism in each entry of the future buffer. When a SETC instruction is executed, the condition value defined by the SETC instruction is saved in the corresponding entry of the condition register and passed on to the future file. Every future buffer entry will use the condition to evaluate the first predicate in the predicate field. If the predicate evaluates to true, the predicate tag is shifted left by two bits and 11 is shifted into the rightmost two bits. A result can be committed if all its predicates are 11, because it means this result is not dependent on any condition. If the predicate evaluates to false, the predicate tag is set to zero, which means this result is squashed.

When the result of an instruction reaches the bottom of the future buffer, the result can be either committed to the register file, discarded, or just held in the future buffer according to its predicate state and valid flag. As shown in Figure 5, if all its predicates in the predicate field

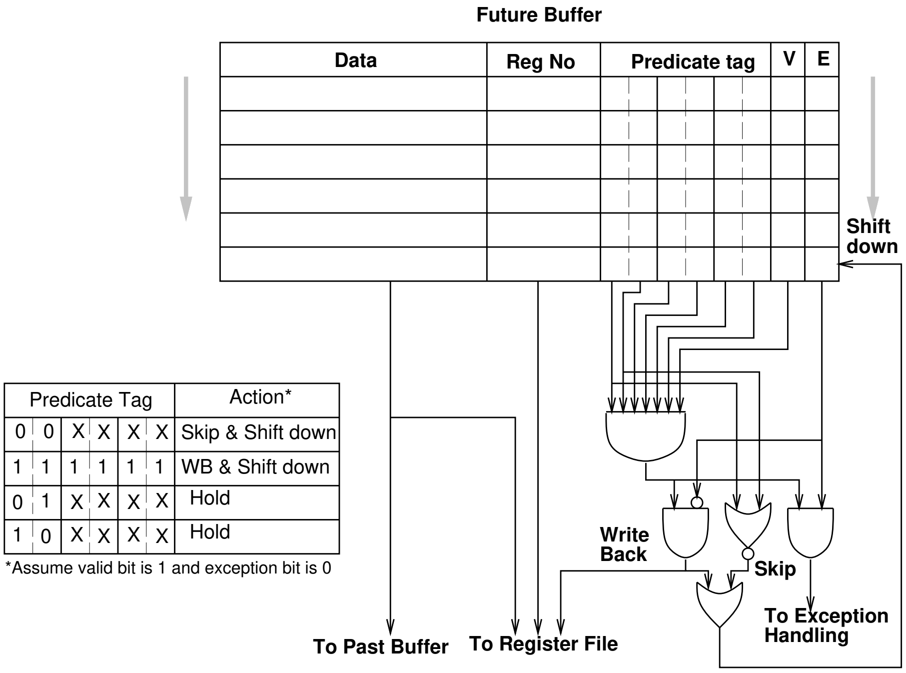


Figure 5: The write back mechanism for future buffer.

are 11 and the valid flag is set, and there is no exception caused by the result, the result can be committed and written back to the register file. If the first predicate in the predicate field is 00, which means this result was squashed, the result is shifted out of the future buffer and discarded. If the predicate field still contains predicates that are dependent on some future conditions, the results are held in the future buffer until all the predicates evaluate to 11 or 00. The committing of a result will also be stalled if its valid bit is false, which means the result is not available yet. When a result is shifted out of the future buffer, regardless of being committed or discarded, the result value is saved in the past buffer for exception recovery. If the result of an instruction is ready to be committed but its exception flag is set, the processor will withhold the write back and signal the exception.

Since our execution model also supports predicated execution, the predicate tag of an instruction may contain predicates that depend on some current conditions. The current conditions are the conditions defined by the previous SETC instructions which are still live in the condition register. In this case, the decoder is responsible for evaluating and updating those predicates before writing them to the predicate field of the corresponding entry in the future buffer. As mentioned in the previous section, the current conditions will be retired when the processor completes the instruction issue of the current basic block and moves to the next basic block. The processor identifies basic block boundaries by a branch instruction, regardless of the branch being executed or not. When the decoder detects a branch instruction, the conditions defined in the current basic block are shifted out of the condition register and passed on to the past-condition register. Note that the condition defined by a SETC instruction issued at the last instruction cycle of the current basic block is regarded as a condition defined in the next basic block and will not be shifted out of the condition register.

Moving an instruction too far above it control dependent SETC instruction may cause a deadlock in the future buffer. A deadlock occurs when the result of an instruction reaches the bottom of the future buffer but cannot be committed or discarded because its dependent condition is not defined yet. In the meantime, the SETC instruction that defines the condition cannot be issued and executed because the future buffer is full. To avoid such a deadlock, the distance between a speculative instruction and its control dependent SETC instructions must be less than the number of entries in the future buffer. In other words, the maximum distance of the speculative execution is limited by the size of the future buffer.

The FIFO mechanism of the future buffer is very similar to the reorder buffer [7, 16] used in superscalar processors. The main disadvantage of a reorder buffer is that it requires associative hardware to provide later instructions with the uncommitted results produced by earlier instructions. The hardware complexity and its associated latency often limit the size of a reorder buffer in superscalar processors [7]. The design of the future buffer eliminates the need of the associative hardware with the assistance of the compiler. Since instruction scheduling is done at compile time, the compiler knows the register dependency and the distance between instructions. If an instruction depends on the result of a previous instruction which may be still in the future buffer when the instruction is issued, the compiler will replace the register name of the operand with an offset which is the distance between the current instruction and the previous instruction. Only operands which depend on the previous results with a distance less than the size of the future buffer need to be renamed with an offset. iF the distances are greater than the size of the future buffer, the instruction can read the results from the register file with the original register names, because the results must have been committed when this instruction is issued. The compile time renaming scheme can greatly simplify the implementation of the future buffer. In other words, the complex associative hardware can be replaced by a simpler address calculation hardware for offset references.

Another advantage of the future buffer is that it allows multiple uncommitted writes to the same register. The multiple uncommitted writes are caused by output dependencies with an instruction distance less than the size of the future buffer, or by the execution of speculative instructions along different control paths which write to the same register. The allocate-on-demand nature of the future buffer allows multiple entries allocated to the same register, and thus allows the compiler to use the buffer more efficiently.

While the future buffer allows multiple uncommitted writes to the same register, there is still a problem when an instruction i needs to read a register which is written by two instructions from different control paths that are merged before the instruction. In this situation, the compiler cannot know which results (offsets) should be used. To solve this problem, we introduce a new instruction, called PHI[5], to select the result from the actual control path and write it to the same register at run time. By inserting the PHI instruction before the instruction i, the compiler can use an offset to reference the result of the PHI instruction and get the correct value for the instruction i. Figure 6 shows an example of how the PHI instruction is used. In our execution model, two control paths will be merged when several basic blocks are fully predicated into a basic block as shown in figure 6. Not that a PHI instruction cannot be moved above the SETC instruction that separates the control paths, because the PHI instruction needs to use its condition value to select the correct result value.

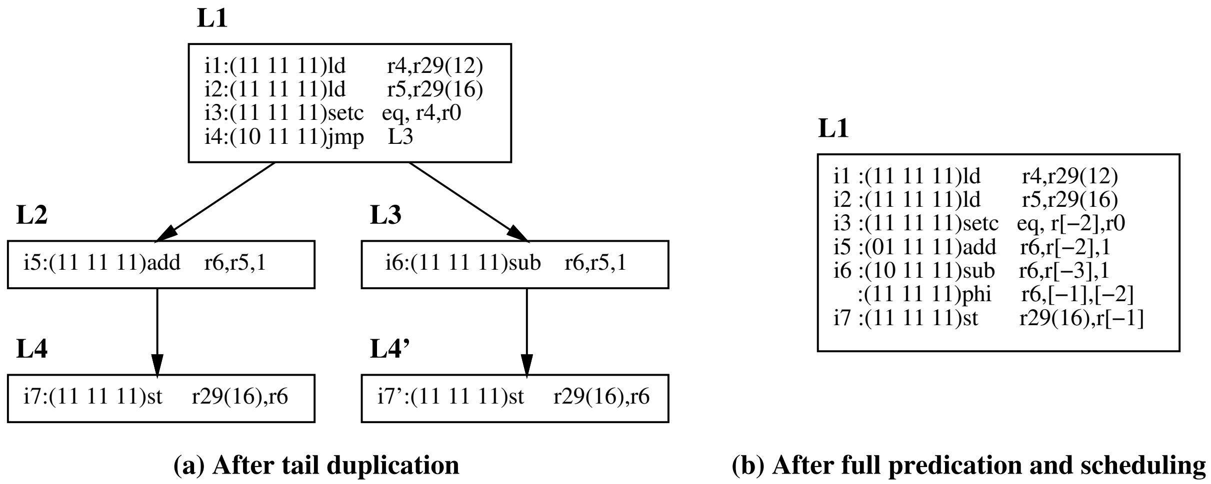


Figure 6: The use of the PHI instruction.

The architecture model also provides a *store buffer* to buffer uncommitted store data. Like the future buffer, the store buffer is organized as a FIFO queue. Each store buffer entry consists of the store address, the store data and its associated predicate tag. The predicate tags are evaluated and updated at run time similar to those in the future buffer. The predicate tag, however, is used for determining the data dependency between a store instruction and a later load instruction rather than for determining if its store data can be committed. The store data can be committed and

[5]We name this instruction PHI because it performs a similar function of the ϕ-function used in the Static Single Assignment (SSA) form [14]

written back to the data cache only when the corresponding entry in the future buffer is committed. Another difference between the store buffer and the future buffer is that the store buffer uses associative hardware to check the data dependencies between load and store instructions. The associative hardware for store buffer will not cost much since the store buffer only needs a small number of entries. When a load instruction is executed, if its load address matches the address field of a store buffer entry and its predicate tag is *dominated* by the predicate tag of the entry, the store buffer will forward the uncommitted store data to the load instruction. A predicate tag P is said to be *dominated* by another predicate tag Q if, for every predicate in the predicates tag Q, the predicate is either 11 or is equal to the corresponding predicate in the predicate tag P.

3.3 Handling Exceptions

With the in-order completion provided by the future buffer, the exception handling and recovery mechanism is very straightforward and simple. When an exception occurs to an instruction, the processor sets the exception flag in the corresponding future buffer entry. The detection of the exception is postponed until the instruction reaches the bottom of the future buffer and is ready to be committed. The logic for detecting the exception is also shown in Figure 5. When the exception is detected and signaled, the processor discards all the data in the future buffer and the store buffer and evokes the exception handling process. The past state stored in the past-condition register and the past buffer as well as the current state stored in the condition register and the register file need to be saved in the memory before the exception handling process starts. The exception handling process will restore these state and data when it returns.

After the exception is handled, if it is not fatal, the processor will restart from the instruction which caused the exception. All the instructions after the excepting instruction must also be re-executed. Two things are needed to ensure the correctness of the re-execution. First, the condition window stored in the condition register needs to be rolled back. During the program execution, the instruction being committed is behind the instruction being decoded with a maximum distance of the size of the future buffer. Because the scope of the condition window is with respect to the instruction being decoded rather than to the instruction being committed, it is possible that some conditions affecting the instructions after the excepting instruction may have been retired and shifted to the past-condition register. Note that the program is restarted from the excepting instruction, hence, these operations must be undone. To roll back the condition register, the recovery process needs to restore the conditions retired after the excepting instruction from the past-condition register back to the condition register. In addition, all the condition values defined by the SETC instructions after the excepting instruction are reset to *undefined*.

Second, the processor must provide the instructions to be re-executed with the register data accessed by offset references. Remember that instructions use offsets to read uncommitted register values from the future buffer. When an instruction is re-executed, the register value which it accesses with an offset may have been committed to the register file or squashed and no longer exists in the future buffer. This problem is solved by the use of the past buffer. The past buffer is implemented as an extension of the future buffer, and can be accessed also by an offset reference. During the program execution, the committed and squashed results being shifted out of the future buffer are all shifted into the past buffer. The number of entries in the past buffer is equal to that in the future buffer. The register values to be accessed by the offset references after exception handling can always be found either in the past buffer or in the future buffer. With the support of the past buffer, the compiler can replace an operand with an offset reference without worrying if the result is committed or not.

3.4 Compiler Support

Our execution model relies on the compiler to exploit ILP across basic block boundaries and to generate the correct code for speculative and predicated execution. In this subsection, we briefly outline the compiler techniques for instruction scheduling and code generation.

Region Formation The region formation algorithm starts from a basic block in the global control flow graph and includes as many basic blocks as possible provided that the basic blocks are dominated by the entry block. The algorithm stops growing a region along a control path when it encounters a function call on the control path. In addition, the algorithm always starts a new region whenever it encounters a loop head (or an entry block of a backward edge), so that there will be no loop formed inside a region. To control possible code expansion caused by the tail duplication, we may limit the levels of conditional branches allowable in a region.

Tail Duplication After the region formation we perform tail duplication. It forces every basic block in a region, except the entry block, to have only one predecessor in the control flow graph. Although the code size may grow in this process, the later full predication pass will eliminate many redundant duplications to reduce the code size.

Branch Conversion In this pass, all conditional branches in a region are converted into a SETC instruction followed by a predicated jump to the branch target. In addition, every instruction is assigned an absolute predicate tag which specifies the control dependencies related to all the SETC instructions on the control path from the entry block to the basic block it is in.

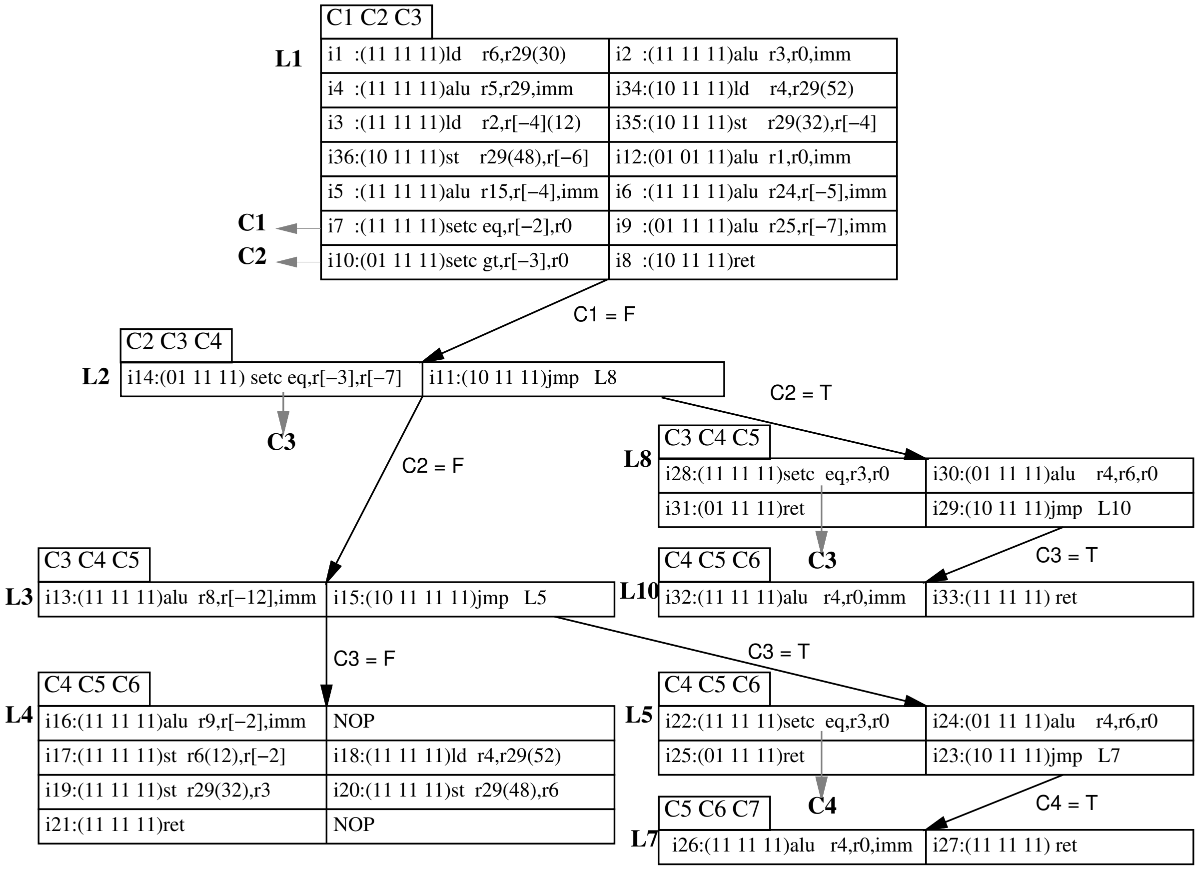


Figure 7: The code segment from *compress* after code generation.

Full Predication In this pass, the compiler traverses the control flow graph of each region to identify candidate blocks for full predicating. If two control paths are different in only a few instructions, the compiler will fully predicate the different parts and eliminate one of the control paths. With full predication, the compiler can reduce the code size as well as eliminate some branch instructions. Full predication, however, may waste resources due to unnecessary execution of instructions in the untaken control paths. In addition, merging two control paths may introduce PHI instructions that may limit the speculative instruction movement. As mentioned earlier, the PHI instructions and the instructions dependent on them cannot be moved above the SETC instruction they are control dependent on. In order to control the overhead in full predication, the compiler uses a threshold to determine whether a candidate basic block should be fully predicated or not. A basic block can be fully predicated only if its size is less than the threshold value.

Instruction Scheduling The compiler performs global instruction scheduling in a region by scheduling basic blocks in a top-down order. When scheduling instructions in a basic block, the compiler only schedules the branch, the SETC instructions, and their dependent instructions. All the other instructions are pushed down to the succeeding basic blocks in order to minimize the required instruction cycles for the basic block. Although pushing those instructions to the succeeding basic blocks may require duplicating the instructions and, thus, increase the code size, we found most of the instructions can be scheduled back to an empty instruction slot in their predecessor blocks in the later scheduling phase, and eliminate their clone instructions. The remaining instructions in the basic block are scheduled using list scheduling.

When scheduling an instruction, the compiler tries to find an empty instruction slot as early as possible along the control path from the entry block. After scheduling a basic block, the compiler selects the next schedulable basic block with or without profile information. A basic block is schedulable if its predecessor has been scheduled. When scheduling with profile information, the basic block with the highest execution count is chosen to be scheduled. If no profile information is available, the schedulable basic block which is closest to the entry block is chosen.

Code Generation After all instructions are scheduled, the compiler generates the predicate tags and the offset references as required. The predicate tag of an instruction is computed by comparing the absolute predicate tag of the instruction with the absolute predicate tag associated with the basic block the instruction is scheduled into. The operand of an instruction is replaced by an offset reference if the operand depends on the result of a previous instruction whose distance is less than the size of the future buffer.

Figure 7 shows the final version of the code shown in Figure 2 generated by the compiler. The machine model for this code is a two-issue processor with 16-entry future buffer and a 6-bit predicate tag. Some basic blocks in the original code have disappeared because all of their instructions have been moved to the predecessor blocks. Also, note some SETC instructions have been moved to the last instruction of their predecessor blocks.

4 Performance Evaluation

Ando et al. [2] have shown that multiple-path speculative execution through predicating outperforms traditional global scheduling techniques and single-path speculative execution techniques such as boosting [19, 18]. Hence, we only compare the performance of the predicate shifting model with the predicating model. Some performance-related issues, which include the size of the future buffer, instruction scheduling with or without profiling, and code expansion rate, are also studied in this section.

4.1 Methodology

We implement the instruction scheduler and simulator based on *pixie* and *xsim* [17]. Pixie is used to partition an executable program into basic blocks and generate dynamic basic block trace for simulation. Xsim is used to construct the global control flow graph. Xsim also provides a disassembler for decoding instructions with given addresses. With the global control flow graph and instructions associated with each basic block, we can perform instruction scheduling and code generation for our machine models. Our simulator simulates the execution of the optimized code cycle-by-cycle, using the dynamic basic block trace generated by pixie. The performance of the optimized program run on a machine model is expressed in terms of the speedup over the MIPS R4000 processor. The execution cycle count of the program run on a R4000 processor is reported by *prof*, which also uses the profile information generated by pixie to derive the execution cycle count.

Program	Dynamic Instructions	R4000 Cycles	R4000 CPI
008.espresso	496M	713M	1.44
022.li	1128M	1722M	1.53
023.eqntott	866M	1406M	1.62
026.compress	87M	126M	1.45
072.sc	148M	223M	1.51
085.gcc	140M	210M	1.49
gawk	164M	230M	1.40
grep	690K	1045K	1.51
cmp	745K	813K	1.09

Table 1: Benchmark Programs and their simulation information.

Table 1 lists the benchmark programs used in our study. It consists of SPEC92 integer benchmark programs *008.espresso*, *022.li*, *023.eqntott*, *026.compress*, *072.sc* and *085.gcc*, and three GNU utilities *gawk*, *grep*, and *cmp*. These programs are compiled into executables using SGI C compiler version 5.3 with *-sopt* (source-to-source optimization) and *-O3* options. The executables are then processed by pixie, xsim, and our code scheduler. Table 1 also lists the dynamic instruction counts and cycle counts reported by prof.

The base machine model used in our simulation is a full 4-issue VLIW processor. There is

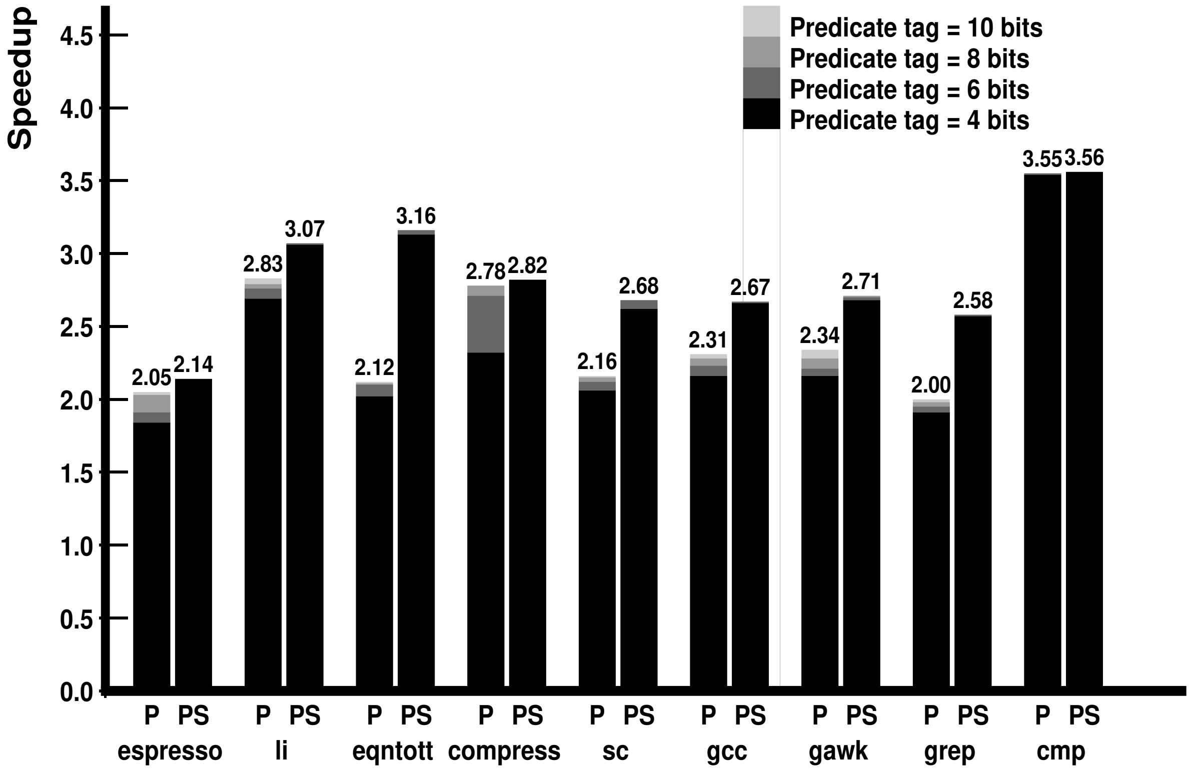


Figure 8: Performance comparison of the predicating model (P) and the predicate shifting model (PS) with predicate tags of different sizes.

no limitation on the combination of instructions that can be issued in each cycle. However, at most one SETC instruction and one branch instruction can be executed in each cycle. If multiple SETC's or branch instructions are scheduled in the same cycle, the scheduler must guarantee that their predicate states are exclusive, i.e., at most one of their predicate tags will evaluate to be true when they are issued. All instructions, except branch instructions, have the same latencies as those of R4000. For branch instructions, we assume a branch target buffer [9] is used and all branch targets except indirect jumps will hit the branch target buffer. Note that branch prediction is not essentialin our model because a conditional branch instruction is always executed at least one cycle after the SETC instruction that defines the branch condition. To simplify our model, we also assume a perfect memory subsystem with a latency of two cycles.

Program	Predicating with predicate tag of				Predicate
	4 bits	6 bits	8 bits	10 bits	Shifting
espresso	8.56	9.75	10.57	11.33	28.16
li	7.44	8.58	9.45	10.13	22.10
eqntott	6.51	7.27	7.88	8.45	13.28
compress	8.45	9.67	10.85	11.69	21.98
sc	7.67	8.85	9.84	10.73	37.41
gcc	7.46	8.70	9.75	10.63	65.03
gawk	9.28	10.99	12.47	13.86	90.07
grep	8.35	9.84	11.10	12.06	104.67
cmp	8.24	9.60	10.69	11.52	22.40
Harmonic Mean	7.96	9.20	10.22	11.07	35.66

Table 2: Average instruction count (static) per region in different predicating models

4.2 Results

Our scheduler and simulator can generate and execute codes for both the predicating model and the predicate shifting model with varying machine configurations. To make the comparison fair, we use the same future buffer mechanism to buffer uncommitted results for both models. In the following performance evaluation results, the default machine model is a 4-issue processor with a 32-entry future buffer and a 8-bit predicate tag. The default threshold value for full predication is zero, and the instruction scheduling is done without using profile information.

Comparison of Predicate Shifting and Predicating Figure 8 shows the performance comparison of the predicate shifting model and the predicating model with predicate tags of different sizes. When a predicate tag size of 10 bits is used for both models, the harmonic mean of speedup is 2.80 for predicate shifting model, and 2.42 for the predicating model. The predicate shifting model achieves 16% improvement over the predicating model. When the predicate tag size is reduceded to 4 bits, the predicate shifting model can still maintain a speedup of 2.79, while the speedup of predicating model drops to 2.25. In this case, The predicate shifting model performs 24% better than the predicating model. The reason for the better performance is that its predicate encoding mechanism does not limit the size of regions. Hence, more ILP can be exploited within larger regions. Table 2 shows the average region sizes of the programs in both models with predicate tags

Program	Number of entries in future buffer			
	16	32	48	64
espresso	2.14 (100%)	2.14 (100%)	2.14 (100%)	2.14
li	3.06 (99.7%)	3.07 (100%)	3.07 (100%)	3.07
eqntott	3.15 (99.7%)	3.16 (100%)	3.16 (100%)	3.16
compress	2.78 (98.6%)	2.82 (100%)	2.82 (100%)	2.82
sc	2.67 (99.6%)	2.68 (100%)	2.68 (100%)	2.68
gcc	2.67 (100%)	2.67 (100%)	2.67 (100%)	2.67
gawk	2.67 (98.5%)	2.71 (100%)	2.71 (100%)	2.71
grep	2.58 (100%)	2.58 (100%)	2.58 (100%)	2.58
cmp	3.56 (100%)	3.56 (100%)	3.56 (100%)	3.56
Harmonic Mean	2.78 (99.3%)	2.80 (100%)	2.80 (100%)	2.80

Table 3: Cycle-count speedup with future buffers of different sizes. (percentages are speedup comparing with 64-entry future buffer)

of different sizes. In the predicating model, the average region size is merely 8 to 11 instructions per region and glows slowly as the size of the predicate tag increases. On the contrary, the region size in the predicate shifting model is independent of the predicate tag size and has 35 instructions per region on average.

Effect of Future Buffer Size on Performance As mentioned in section 3.2, the maximum distance of speculative execution is limited by the size of the future buffer. With a larger future buffer, the compiler can move an instruction further above its dependent SETC instructions. However, a large future buffer is costly because it requires the same number of entries in the past buffer and, most importantly, it requires additional bits in every instruction word to specify offset references for accessing results in the future buffer. Table 3 shows the cycle-count speedup of the 4-issue machine model with future buffers of 16, 32, 48, and 64 entries. The result shows that a future buffer with 32 entries can perform as well as a future buffer of 64 entries. The result also shows that the performance of a 16-entry future buffer can be within 99.3% of a 64-entry future buffer. Hence, the size of the future buffer need not be very large.

Instruction Scheduling with Profile Information The performance of instruction scheduling with and without profile information is compared in Table 4. It can be seen that profile information

Program	Speedup	
	Without profiling	With profiling
espresso	2.14	2.14 (100.00%)
li	3.07	3.11 (101.30%)
eqntott	3.16	3.17 (100.32%)
compress	2.82	2.82 (100.00%)
sc	2.68	2.69 (100.37%)
gcc	2.67	2.70 (101.11%)
gawk	2.71	2.72 (100.36%)
grep	2.58	2.63 (101.93%)
cmp	3.56	3.56 (100.00%)
Harmonic Mean	2.80	2.81 (100.36%)

Table 4: Cycle-count speedup comparison of instruction scheduling with and without profileinformation.

can only improve performance by an average of 0.36%. Profiling information is very important in traditional global instruction scheduling techniques such as trace scheduling [5], and in single-path speculative execution supported by boosting. They rely on the profile information to select the most-frequently executed control path for performing speculative instruction movement. Optimizing one control path, however, may affect the performance of the other control paths. In our execution model, all control paths can be optimized through multiple-path speculative execution provided that there are sufficient resources. As a result, our execution model can achieve good performance without profile information.

Code Expansion Rate and Full Predication To support the predicate shifting execution model, the scheduler needs to perform tail duplication. Full tail duplication may cause the code size to grow exponentially. As mentioned in Section 3.4, we can reduce the code size by limiting the level of conditional branches in a region and by performing full predication to merge control paths. In our region formation pass, the maximum level of conditional branches in a region is 16. In the full predication pass, the compiler uses a threshold to determine whether a basic block should be fully predicated. With a larger threshold value, more basic blocks can be fully predicated and the code size will be reduced. On the other hand, if the threshold value is too large, the performance may suffer as discussed in Section 3.4. Table 5 shows the effects of the threshold value on the

Program	Threshold for fully predicating							
	0		4		8		16	
	Expansion	Speedup	Expansion	Speedup	Expansion	Speedup	Expansion	Speedup
espresso	2.93	2.14	2.40	2.14	1.58	2.07	1.46	2.07
li	2.37	3.07	1.88	3.07	1.80	3.07	1.77	3.07
eqntott	1.76	3.16	1.45	3.23	1.39	3.23	1.36	3.29
compress	2.23	2.82	1.72	2.76	1.58	2.73	1.52	2.73
sc	3.66	2.68	2.11	2.66	1.93	2.64	1.79	2.64
gcc	6.07	2.67	2.67	2.63	2.28	2.62	2.13	2.59
gawk	7.30	2.71	2.41	2.68	2.11	2.66	2.05	2.64
grep	9.17	2.58	1.98	2.59	1.79	2.58	1.72	2.58
cmp	2.26	3.56	1.74	3.56	1.59	3.56	1.54	3.56
Harmonic Mean	3.56	2.80	2.01	2.79	1.76	2.77	1.69	2.77

Table 5: The effects of full predicating on code expansion rate and speedup

performance and the code expansion rate. The code expansion rate is the ratio of the code size after full predication pass and the original code size. When the threshold value is 0, which means no basic block is fully predicated, the harmonic mean of the code expansion rates is 3.56. When the full predication is performed, the code expansion rate is reduced to 2.01 with the threshold value of 4, and to 1.69 with the threshold value of 16. The performance doesn't vary much with the threshold values used in Table 5. This result shows that we can reduce the code expansion rate to a reasonable range without harming the performance.

5 Conclusions

This paper presents a control dependency encoding and manipulating mechanism, called predicate shifting, to effectively support both predicated and speculative execution. The predicate shifting mechanism provides the compiler and the processor a cost-effective way to specify and to store the control dependencies of an instruction. The key idea is using a shifting condition/predicate window to specify the scope of branch. With the support of predicate shifting mechanism, the compiler can fully predicate basic blocks or speculatively move instructions from multiple control paths above the conditional branches they are dependent on. Unlike the previous predicating mechanism [2], this mechanism will not limit the number of conditions in a region and thus can achieve good performance with a small predicate tag. The simulation results show that the predicate shifting model can achieve 16% performance improvement over the predicating model when using a 10-bit predicate tag, and achieve 24% performance improvement when using a 4-bit predicate tag. The experimental results also shows that, with the support for multiple-path speculative execution, the compiler can effectively exploit ILP across basic block boundaries without relying on profile information. This is a significant improvement over other schemes [5, 3, 18], which can only speculate along one selected control path.

This paper also presents a structure, called future buffer, to buffer uncommitted results and to evaluate the predicate state associated with each result. The FIFO nature of the future buffer can simplify exception handling and allow multiple uncommitted writes to the same register. To avoid complex hardware for associative lookup, we introduce an offset reference mechanism to access uncommitted results in the future buffer. The experimental results show that a future buffer of 16 entries is able to provide sufficient buffering space for a 4-issue processor.

One of the possible drawbacks of the predicate shifting mechanism is the code expansion caused by tail duplication. We solve this problem by limiting the level of conditional branches in a region and by using full predication to merge control paths. With these methods, we can reduce the code expansion rate to a reasonable factor without harming the performance.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Weslay Publishing Company, Massachusetts, 1986.

[2] H. Ando, C. Nakanishi, T. Hara, and M. Nakaya. Unconstrained Speculative Execution with Predicated State Buffering. In *Proceedings of the 22th International Symposium on Computer Architecture*, pages 126–137, June 1995.

[3] P. O. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter, and W. W. Hwu. IMPACT: An Architectural Framework for Multiple-Instruction-Issue Processors. In *Proceedings of the 18th International Symposium on Computer Architecture*, pages 266–275, May 1991.

[4] R. P. Colwell, R. P. Nix, J. J. O'Donnell, D. B. Papworth, and P. K. Rodman. A VLIW Architecture for a Trace Scheduling Compiler. In *Proceedings of Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 180–192, October 1987.

[5] J. A. Fisher. Trace Scheduling: A Technique for Global Microcode Compaction. *IEEE Transactions on Computers*, C-30(7):478–490, July 1981.

[6] P. Y. T. Hsu and E. S. Davidson. Highly Concurrent Scalar Processing. In *Proceedings of the 13th International Symposium on Computer Architecture*, pages 386–395, June 1986.

[7] Mike Johnson. *Superscalar Microprocessor Design*. Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[8] M. S. Lam and R. P. Wilson. Limits of Control Flow on Parallelism. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 46–57, June 1992.

[9] J. K. F. Lee and A. J. Smith. Branch Prediction Strategies and Branch Target Buffer Design. *IEEE Computer*, 17(1):6–22, January 1984.

[10] S. A. Mahlke, W. Y. Chen, B. R. Rau W. W. Hwu, and M. S. Schlansker. Sentinel Scheduling for VLIW and Superscalar Processors. In *Proceedings of Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 238–247, October 1992.

[11] S. A. Mahlke, R. E. Hank D. C. Lin, W. Y. Chen, and R. A. Bringmann. Effective Compiler Support for Predicated Execution Using the Hyperblock. In *Proceedings of MICRO-25*, pages 45–54, December 1992.

[12] S. A. Mahlke, R. E. Hank, J. E. McCormick, D. I. August, and W. W. Hwu. A Comparison of Full and Partial Predicated Execution Support for ILP Processors. In *Proceedings of the 22th International Symposium on Computer Architecture*, pages 138–149, June 1995.

[13] D. N. Pnevmatikatos and G. S. Sohi. Guarded Execution and Branch Prediction in Dynamic ILP Processors. In *Proceedings of the 21th International Symposium on Computer Architecture*, pages 120–129, April 1994.

[14] J. Ferrante R. Cytron and B. K. Rosen. Efficiently Computing Static Single assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, Oct. 1991.

[15] B. Ramakrishna Rau, D. W. L. Yen, W. Yen, and R. A. Towle. The Cydra 5 departmental Supercomputer. *IEEE Computer*, 22(1):12–35, January 1989.

[16] J. E. Smith and A. R. Pleszkun. Implementation of Precise Interrupts in Pipelined Processors. In *Proceedings of the 12th International Symposium on Computer Architecture*, pages 36–44, June 1985.

[17] M. D. Smith. Tracing with pixie. Technical report, Stanford University, Stanford, California 94305, November 1991. Technical Report CSL-TR-91-497.

[18] M. D. Smith, M. A. Horowitz, and M. S. Lam. Efficient Superscalar Performance Through Boosting. In *Proceedings of Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 248–259, October 1992.

[19] M. D. Smith, M. S. Lam, and M. A. Horowitz. Boosting Beyond Static Scheduling in a Superscalar Processor. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 344–455, May 1990.

[20] M. Srinivas, A. Nicolau, and V. H. Allan. An Approach to Combine Predicated/Speculative Execution for Programs with Unpredictable Branches. In *Proceedings of the IFIP WG10.3 Working Conference on Parallel Architectures and Compilation Techniques*, pages 147–156, August 1994.

[21] D. W. Wall. Limits of Instruction-Level parallelism. In *Proceedings of Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 176–188, April 1991.