Unconstrained Speculative Execution with Predicated State Buffering

Hideki Ando, Chikako Nakanishi, Tetsuya Hara, Masao Nakaya
System LSI Laboratory
Mitsubishi Electric Corporation
4-1 Mizuhara, Itami, Hyogo, 664 Japan

Abstract

Speculative execution is execution of instructions before it is known whether these instructions should be executed. Compiler-based speculative execution has the potential to achieve both a high instruction per cycle rate and high clock rate. Pure compiler-based approaches, however, have greatly limited instruction scheduling due to a limited ability to handle side effects of speculative execution. Significant performance improvement is, thus, difficult in non-numerical applications. This paper proposes a new architectural mechanism, called predicating, which provides unconstrained speculative execution. Predicating removes restrictions which limit the compiler's ability to schedule instructions. Through our hardware support, the compiler is allowed to move instructions past multiple basic block boundaries from any succeeding control path. Predicating buffers the side effects of speculative execution with its predicate, and the buffered predicate efficiently commits or squashes the side effects. The mechanism also provides a speculative exception handling scheme. The scheme, called the future condition, properly postpones speculative exceptions and efficiently restarts the process. We show that our mechanism can be implemented through a modest amount of hardware with little complexity. The evaluation results show that our mechanism significantly improves performance, and achieves a 2.45x speedup over scalar machines.

1 Introduction

Limit studies of available instruction-level parallelism (ILP) [10][20] show that, in non-numerical applications, a basic block has a very limited amount of ILP. Thus just adding extra function units does not necessarily improve performance. The primary reason of this limitation is existence of control dependence constraints. The limit studies show that the amount of available ILP dramatically increases if the control dependences are eliminated. *Speculative execution* is execution of instructions before it is known whether these instructions should be executed. For good performance, control-dependent instructions must be executed before their dependences are resolved.

Most state-of-the-art microprocessors exploit ILP through superscalar techniques [14][16][19]. Although superscalar machines exhibit a good instruction per cycle (IPC) rate, complicated hardware is required since they rely on dynamic instruction scheduling through hardware to exploit ILP. This complication imposes penalties on both hardware amount and cycle time. Furthermore, a run-time scheduler is unable to achieve sophisticated instruction scheduling due to complexity limits.

Very long instruction word (VLIW) machines, on the other hand, can potentially overcome these problems. In VLIW machines, instruction scheduling is optimized by the compiler, and consequently they need only simple hardware. The compiler fundamentally has the ability to optimize schedule code through analyzing critical paths from a large window of instructions and using sophisticated instruction heuristics for scheduling. Yet, pure compiler-based approaches [7][15] to speculative execution have greatly limited instruction scheduling due to limited ability to handle side effects of speculative execution.

Thus some hardware support for side effect handling is necessary to make the best use of the compiler's scheduling ability. Many mechanisms have been proposed (e.g. guarding [8] and boosting [17]). They reduce constraints in speculative code motion, but still do not allow the compiler to have enough freedom for scheduling.

In this paper, we propose a cost-effective architectural support for unconstrained speculative execution. Our mechanism, referred to as *predicating*, introduces the optimal combination of predication and speculation. An instruction is predicated with its control-dependent branch conditions. The side effects caused by speculative execution are buffered with its predicate. This buffered predicate efficiently commits or squashes the side effects and appropriately handles exceptions caused by the speculative execution. Predication allows the simple in-order issue machine to execute instructions in multiple basic blocks simultaneously, and the predicated state buffering mechanism provides the compiler with unconstrained speculative code motions. Section 2 reviews speculative execution and related work. Section 3 proposes the predicating architecture. In this section, we also briefly describe our instruction scheduling algorithm. Section 4 shows evaluation results. Finally, Section 5 concludes this paper.

ISCA'95, Santa Margherita Ligure Italy

2 Background

Three types of constraints exist in exploiting ILP: resource conflicts, data dependences, and control dependences. Reducing constraints is necessary to increase exploitable ILP. Speculative execution is a technique to remove control dependences among these constraints. In this section, we will review issues in speculative execution, and then describe existing architectural techniques.

2.1 Speculative Execution

An instruction is said to be control-dependent upon a conditional branch if it is unknown whether that instruction should be executed or not unless that branch is determined. Consider the following example:

```
i1:    if (r1)
i2:        r2 = load r3;
       else
i3:        r4 = r2 + r3;
```

We do not know whether instruction `i2` or `i3` should be executed or not until branch instruction `i1` is executed. Speculative execution allows instruction `i2` and/or `i3` to be executed before the execution of branch instruction `i1` is completed. In static instruction scheduling, the compiler moves instruction `i2` and/or `i3` above branch instruction `i1` for speculative execution.

There are two problems in the speculative code motions. The first problem is to preserve program semantics. A code motion is said to be *illegal* if the moved operation changes the original program semantics. In the example above, the code motion of instruction `i2` above instruction `i1` is illegal because instruction `i2` overwrites register `r2` whose previous value is necessary for instruction `i3`. Similarly, an illegal code motion exists on memory locations. The compiler can transform the illegal code motion on a register to legal through *register renaming*. In register renaming, the compiler assigns a register which is not live on the side-effects causing path as the destination register. The compiler then inserts an instruction which copies the value from the newly assigned register to the original destination register. In the example above, the compiler can speculatively move branch instruction `i2` above instruction `i1` through register renaming as the following:

```
i2':   r5 = load r3;
i1 :   if (r1)
i4 :       r2 = r5;
       else
i3 :       r4 = r2 + r3;
```

where register `r5` is not live on the path to the ELSE part. The destination register of instruction `i2` is renamed into a dead register (`r5`) in instruction `i2'`, and copy instruction `i4` which copies the result into the original register is inserted. This renaming technique, however, cannot be applied to illegal code motion on a memory location.

The second problem is to handle an exception caused by a speculative operation. A code motion is said to be *unsafe* if the moved operation may cause an exception. In the example above, the code motion of instruction `i2` above branch instruction `i1` is unsafe because the load instruction may cause an exception. An exception caused by a speculative instruction is termed a *speculative exception*. If a speculative exception occurs, immediate handling like non-speculative exception handling incorrectly terminates the program or decreases performance because it is unknown whether that handling is necessary or not. Thus, handling of speculative exceptions should be postponed until the result of the excepting instruction is found to be really necessary. Although a compiler can transform an illegal code motion to legal, a compiler cannot transform an unsafe code motion to safe. Therefore, a compiler must conservatively schedule instructions which may cause an exception.

Since exceptions occur infrequently, one may think that the adverse effect of unsafe code motions is negligibly small. This is not the case in speculative execution. Suppose that a load instruction dereferences a pointer to a next element in a loop program which traverses a linked list. If the load instruction is speculatively executed, it attempts to dereference a NULL pointer in the last iteration, and thus an exception occurs. This type of speculative code motion is quite effective for performance improvement because dereferences are often in a critical path. As a result, aggressive unsafe code motions considerably increase the frequency of exceptions.

Besides postponing handling, there is one more requirement for the handling of speculative exceptions: restarting the process. That is, if a caused speculative exception is non-fatal, the process should be restarted after the handling. This restart problem includes two difficult problems. The first problem is to select instructions which must be re-executed. Just re-execution of the excepting instruction is not sufficient. The speculative instructions which are directly or indirectly dependent upon the excepting instruction must be re-executed since they used polluted operands. This re-execution of the speculative instructions is termed *recovery* from a speculative exception. The second problem exists in the recovery process. To preserve the program semantics, all of the operands of the re-execution instructions in the recovery process must be available. That is, if the compiler moves an unsafe instruction, operand registers of succeeding instructions which may be re-executed must be live until the commit point of that unsafe instruction. This increases the number of live registers, and thus puts pressure on the compiler register allocation.

For more understanding of these problems, consider the following program segment:

```
i1':   r1 = load r2;
i2 :   r3 = r3 + 1;
i3':   r4 = r1 + r5;
i4':   r6 = r4 & 1;
i5 :   branch LAB if (r3)
```

where instructions `i1'`, `i3'`, and `i4'` are speculative instructions upon branch instruction `i5`, while instruction `i2` is a non-speculative instruction. If speculative instruction `i1'` causes an exception, exception handling must be postponed until branch instruction `i5` is executed. In other words, the handling of the exception must be postponed until the exception is committed. If the exception is committed, the exception is handled; otherwise the exception is squashed. Since instruction `i3'` used corrupted register `r1` and instruction `i4'` used the corrupted result of instruction `i3'`, they must be re-executed. Instruction `i2` must not be re-executed because the re-execution of instruction `i2` destroys the semantics. The compiler must not re-allocate registers `r2` and `r5` until the exception commit point even though no instructions refer

to the value in these registers because these values may be used in the re-execution.

2.2 Existing Mechanisms

The guarded instruction architecture [8] predicates a branch-dependent instruction with its dependent branch condition. The predicated instruction is referred to as a *guarded instruction*. Guarded instructions are issued and executed even if the value of the predicate is unknown. Before the instruction reaches the write-back stage in the pipeline, the value of the predicate is specified. If the predicate evaluates to true, the result is committed; otherwise the result is squashed. The compiler must schedule guarded instructions so that its predicate will be specified before the instruction writes the result. Although the guarded instruction architecture allows speculative execution, it is severely limited. That is, speculative state is allowed to live only in the pipeline. Control dependences still exist in the form of data dependences in terms of predicate operand availability. We term this type of speculative execution *squashing speculation*. Squashing speculation is found in a number of work [2][6][13].

The non-excepting instruction architecture [5] supports the handling of speculative exceptions. It labels an unsafe instruction as a *non-excepting instruction*. A non-excepting instruction has the same operation as the normal instruction associated with the non-excepting instruction, but never signals an exception. If the non-excepting instruction causes an exception, it just completes the execution and marks the result to indicate that instruction caused an exception. If a normal instruction uses the polluted result later, the hardware signals the exception. The non-excepting architecture provides a solution for postponing speculative exception handling. Yet, the recovery problem still remains. First, the original excepting address is unknown when the hardware detects the postponed speculative exception. Second, it is difficult to select instructions which must be re-executed, and the availability of every operand of instructions which are re-executed cannot be guaranteed.

The sentinel scheduling architecture [12] extends the non-excepting instruction architecture to solve the problem stated above. The sentinel scheduling architecture labels every speculative instruction with a *speculative modifier*. If an unsafe instruction causes a speculative exception, the excepting instruction writes the excepting address into the destination register, instead of writing a polluted result, and marks the register. If some time later a speculative instruction refers to the marked register, it just copies the excepting address and mark into its destination register. The speculative exception is detected if a non-speculative instruction or some explicit special instruction refers to the marked register. This checking instruction is called a *sentinel*. If the exception is detected, the machine identifies the original excepting instruction by the excepting address stored in the marked register. The recovery is performed by re-executing all speculative instructions from the excepting point until the exception commit point. Although the sentinel scheduling architecture provides a good solution for postponing speculative exceptions and specifying the original excepting instruction, the recovery problem still partly remains. Operand availability of re-execution instructions must be preserved by the compiler's register allocation. Bringmann et al. [3] have proposed an architectural technique for this recovery problem in a specific scheduling model (*superblock scheduling* [4]) where speculative code motions are allowed in a single likely path. This technique, however, does not provide a solution for the general scheduling model which allows code motions from both paths of a branch.

Boosting [17] provides unconstrained speculative code motions in a *trace*, a predicted single path of control flow. Boosting labels a speculative instruction with the number of its dependent branches. The result of speculative execution is buffered in a shadow register. If all dependent branches of a speculative instruction are found to be correctly predicted, the buffered result is committed; otherwise, the result is squashed. Boosting also properly handles speculative exceptions with a compiler assist. A one-bit shifter records outstanding speculative exceptions. That is, if a speculative exception occurs, the machine sets a location in the shifter, which corresponds to the number of dependent branches of the instruction. If a branch is found to be correctly predicted, the shifter is shifted. If the bit is popped out from the shifter, then the speculative exception is detected. If a speculative exception is detected (or committed), all data in shadow registers are discarded, and the machine then calls an exception handler. The handler refers to a jump table with the address of the exception commit point. Each entry of the jump table points to a *recovery code* [18] associated with each commit point. The compiler generates the recovery code which contains instructions the machine needs to re-execute for each commit point. The machine executes the recovery code in user mode. During the execution, the original exception occurs again. After the handling the exception, the recovery code completes by jumping to the predicted target of the branch which committed the exception. Boosting provides a good support for unconstrained speculative scheduling. Yet, scheduling is still limited in a trace which is a single control flow. For the speculative exception handling problem, the recovery code and the jump table double the size of the original code.

3 Predicating

In this section, we propose an architectural mechanism called *predicating*. Predicating provides the compiler with unconstrained speculative code motions. Specifically, the compiler is allowed to move instructions up from any path across multiple basic block boundaries. This ability is quite effective for non-numerical applications where branches are not predictable. In the following subsections, we first describe our execution model and mechanism. We then briefly describe our instruction scheduling algorithm. The mechanism of handling speculative exceptions is discussed next.

3.1 Execution Model

Our execution model has two machine states: a *sequential state* and *speculative state*. The sequential state consists of the results of instructions whose control dependencies are all resolved; the speculative state consists of the results of instructions which have at least one unresolved control dependence. The result in the speculative state has its predicate which is a commit condition to the sequential state.

The instruction is predicated with its control-dependent conditions. At the issue point, the predicate of the instruction is evaluated. If the predicate evaluates to true, the instruction is executed and writes the result into the sequential state. This is non-speculative execution. If the predicate evaluates to false, the instruction is simply squashed because the execution is no longer necessary. In the last case, at least one branch condition of the predicate is not determined, that is, the value of the predicate is not *specified*. In this

case, the instruction is executed, but writes the result into the speculative state. Unlike non-speculative writes, those speculative writes label the result with the predicate for later commit. The predicate of the result in the speculative state is evaluated every cycle by referring to the branch conditions. During cycles in which the predicate is unspecified, the result is held continuously. If the predicate evaluates to true, the result is committed; if the predicate evaluates to false, the result is squashed.

If an instruction causes a speculative exception, the exception is not handled immediately but the instruction simply writes the corrupted result into the speculative state. When the instruction writes the corrupted result, the instruction also marks the result to indicate that the result is corrupted. Since this write is speculative, the corrupted result is predicated like non-corrupted results. In other words, outstanding exceptions are buffered with the predicate. If later the predicate of the outstanding exception evaluates to true, hardware detects the exception. The machine then starts to recover the corrupted machine state through re-execution of instructions. During the re-execution, the original exception occurs again, and this time it is handled. The re-execution is completed when the re-execution reaches the original exception detected point.

3.2 Architecture

An instruction in the predicating architecture has the following format:

```
predicate ? operation
```

Semantically, the result of the execution specified by the operation part is valid if and only if the commit condition specified by the predicate part is true.

Figure 1 illustrates the organization of the predicating architecture with *N*-instruction issue. The architecture differentiates itself from conventional VLIW machines by including a control path, predicated register file, and predicated store buffer.

The control path evaluates the predicate of instructions which are executed in the datapath by referring to the branch conditions stored in the condition code register (CCR). If the control path answers "true," the result of the execution in the datapath is committed; if the control path answers "false," the result of the execution in the datapath is squashed.

The register file consists of sequential registers and shadow registers. The sequential register contains the sequential state; the shadow register contains the speculative state. If the predicate evaluates to true in the control path, the result is written into the sequential register; if the predicate evaluates to false in the control path, the write of the result is squashed; and if the predicate evaluates to an unspecified value, the result is written into the shadow register. An instruction explicitly specifies whether it should fetch operands from the sequential registers or the shadow registers. Unlike the shadow register in boosting [17], the shadow register in predicating has full control dependence information associated with the stored data. When an instruction writes the result into the shadow register, the instruction also writes its predicate into the predicate storage associated with the destination register.

Theoretically, multiple shadow registers for each sequential register are required because each sequential register can potentially have a number of speculative states which have a different

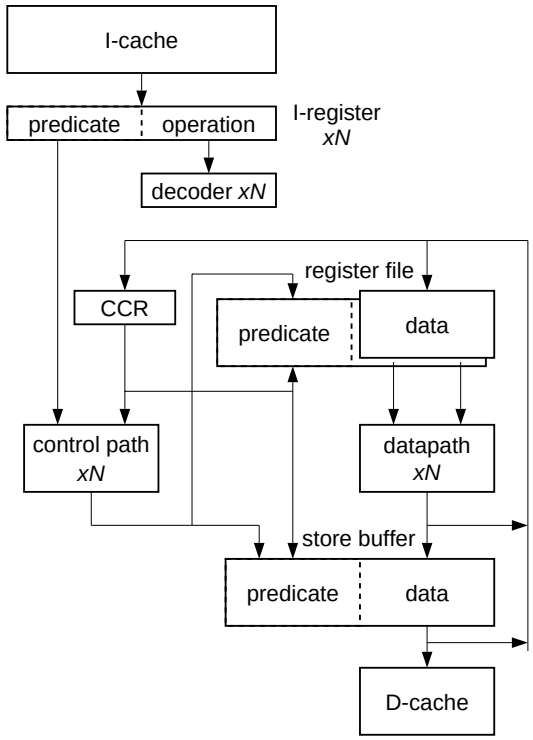

Figure 1: Hardware organization (*N*-instruction issue)

predicate. In order to keep the cost reasonable, only a single shadow register is provided for each sequential register. This may cause a storage conflict among the writes of the results with different predicates, but this conflict has little impact on performance because it rarely occurs[1]. Figure 2 shows the configuration of our register file. Each entry of the register file contains two values, one predicate, and three flags. One of two data storages stores speculative data, and the other stores non-speculative data. Flag W indicates which storage stores speculative data. Flag V indicates that the speculative data indicated by flag W is valid. Flag E indicates that there exists an outstanding speculative exception.

The result of speculative execution is written into storage indicated by flag W of the destination entry. At the same time, flag V is set and the predicate is written. Each entry of the register file has dedicated hardware which evaluates the predicate stored in its entry. The commit action is done by updating flags V and W according to the result of the evaluation. If the predicate evaluates to true, flag W is flipped and the flag V is reset. This action commits the speculative data. If the predicate evaluates to false, flag V is reset. This action squashes the speculative data.

So far, we have assumed that the predicate can be a general form of a boolean expression. The hardware required to evaluate a general predicate is obviously unacceptable in both hardware amount and signal delay time. To reduce these costs to a reasonable level, we limit the expression to an ANDed operation with negation. For example, we allow predicates `c1&c2` or `c1&!c2`, but do not allow

1. Our evaluation shows that a single shadow register model lowers just 0 - 1% performance under an infinite shadow register model.

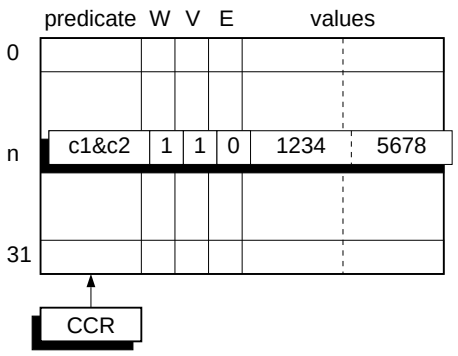

Figure 2: Predicated register file

`c1|c2` (a branch condition has a name `c`*n*, where *n* specifies the *n*-th entry of CCR). We encode the predicate in a vector where each entry is associated with a branch condition. Each entry has a boolean value which is necessary to let the predicate be true. For example, suppose CCR has three entries to hold three branch conditions `c1`, `c2`, and `c3`. We encode a predicate `c1&!c2&c3` to `{1,0,1}`. Since we allow a value of "don't care" for a condition, a predicate `c1&c3` is encoded to `{1,X,1}`. The restriction in the predicate expression makes the evaluation of a predicate quite simple. Intuitively, just a match operation between the encoded predicate and the content in CCR evaluates the predicate. For example, if CCR holds `{1,0,1}`, a predicate `c1&!c2&c3` evaluates to true because the predicate is encoded to the same vector. Since we allow a value of "don't care" for a branch condition in a predicate, the associated branch condition is masked in the match operation. In addition, the predicate evaluating hardware checks whether the unmasked branch condition has an unspecified value or not. If at least one unmasked branch condition is not specified, the predicate evaluates to an unspecified value regardless of the match operation result.

The store buffer is organized as a FIFO buffer, and both speculative and non-speculative data are buffered before the D-cache write. Like the register file, the data in an entry is predicated and the predicate has hardware to evaluate itself. Each entry also has three flags: W, V, and E. Flag W indicates that the data in that entry is speculative; flag V indicates the data is valid; and flag E indicates that there exists an outstanding speculative exception. When an instruction writes data into the tail of the store buffer, flag V is set. If a store instruction is speculative, that is, the predicate evaluates to an unspecified value in the control path, flag W is set when the instruction appends the data. If the data in the head of the store buffer is valid and non-speculative, the data is written into the D-cache. If the predicate in an entry evaluates to true, the data in that entry is committed, that is, flag W is reset; if the predicate in an entry evaluates to false, the data in that entry is squashed, that is, flag V is reset.

3.3 Instruction Scheduling

The instruction scheduling scheme is strongly related to the predicating mechanism. To further explain our mechanism, we describe our instruction scheduling algorithm. Since a comprehensive description of the scheduling algorithm is beyond the scope of this paper, we describe only what is related to our mechanism.

In the scheduling for predicating architecture, the compiler groups some basic blocks, referred to as a *region* [1], for predicated execution. Our scheduler basically moves instructions within the region. A region is a control flow graph (CFG) which includes a header block, and the header block dominates all other blocks in the region. The header block is the only entry, and there exist one or more exits. Our region selection algorithm grows a region (candidate) from a seed block (usually a loop head) which is an initial region in the direction of the edges in the CFG. The region is grown if the growth to the next block of the current region is considered beneficial. We use a heuristics which is a function of static branch predication to drive this region growth. After choosing blocks as the region, the compiler duplicates the blocks (if necessary) so that the header is able to dominate them.

The control transfer within the region is removed by predicating instructions. Thus, once the control is transferred to the top of a region, the machine sequentially executes instructions in the region. If the condition for exiting the current region is met, the control is transferred from the middle or bottom of the region to a target region.

Within a region, the compiler has no restriction in terms of speculative code motions. Although we implement code motions across region boundaries in some limited form, code motions are basically limited within a region. This greatly simplifies the instruction scheduling and reduces the cost of calculating instruction availability. Furthermore, this limitation implies that the speculative state built in a region depends only upon branch conditions which are specified in the current region. That is, the speculative state is never live beyond the current region, and is definitely committed or squashed in the current region. With this property, the speculative state is said to be *closed* in the region. This property has significant benefits in handling speculative exceptions. We will discuss these further in Section 3.5. Since the speculative state is closed in a region, all branch conditions are reset to an unspecified value by the hardware on an exit from the current region.

As described in the previous subsection, we limit the expression of a predicate to an ANDed operation. This limits the CFG of a region. Obviously, if any block in a region has only a single path from the header block, the predicate limitation is satisfied. If there exists a join block which has multiple paths from the header block, and if the join block has an *equivalent*[2] block which has a single path from the header block, then the region is also subject to the predicate limitation since the control dependence of the join block is the same as the control dependence of the equivalent block. Otherwise, the join block is duplicated so that the region is able to be subjected to the predicate limitation.

3.4 Example

We will now present a small example. Figure 3 shows a scalar code before scheduling. Jump instructions for both branch paths are inserted for explanation. Figure 4 lists the scheduled code of the region hatched in Figure 3 for a 2-issue machine. The suffix `.s` of a register in an instruction specifies a shadow storage of the register. The suffix for a destination register is assigned just for convenience since the control path assigns it at run-time, and thus

2. Block X is said to be equivalent to block Y if block X dominates block Y and block Y post-dominates block X.

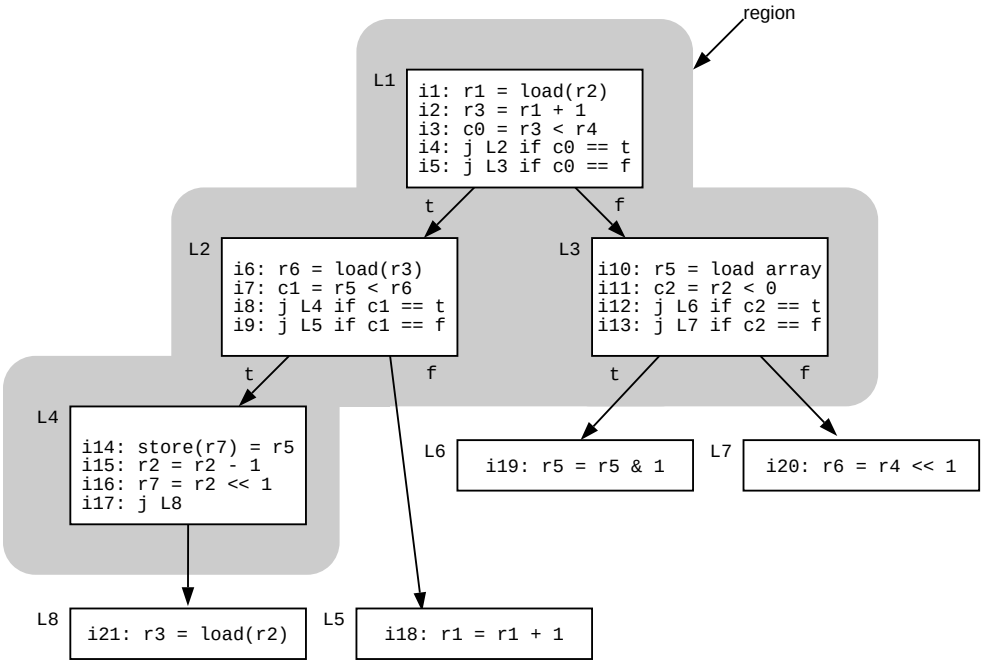

Figure 3: Scalar code example

```
(1)   i1 :   alw      ?  r1 = load(r2);        i15:   c0&c1    ?  r2.s = r2 - 1;
(2)   i10:   !c0      ?  r5.s = load array;    i14:   c0&c1    ?  store(r7) = r5;
(3)   i2 :   alw      ?  r3 = r1 + 1;          i16:   c0&c1    ?  r7.s = r2.s << 1;
(4)   i6 :   c0       ?  r6 = load(r3);        i3 :   alw      ?  c0 = r3 < r4;
(5)   i11:   alw      ?  c2 = r2 < 0;          -  :   alw      ?  nop;
(6)   i7 :   alw      ?  c1 = r5 < r6;         i12:   !c0&c2   ?  j L6;
(7)   i9 :   c0&!c1   ?  j L5;                 i17:   c0&c1    ?  j L8;
(8)   i13:   !c0&!c2  ?  j L7;                 -  :   alw      ?  nop;
```

Figure 4: Scheduled code for two-issue machine

Table 1: Machine state transition

cycle	sequential state write	speculative state						CCR		
		write				commit	squash	c0	c1	c2
1		c0&c1	r2							
2	r1	c0&c1	sb1							
3	r3	!c0	r5	c0&c1	r7					
4								T		
5	r6						r5			F
6									T	
7						r2,r7 sb1				

it is not encoded into an instruction word. The latency of a load instruction is two cycles; the latency of all other instructions is one cycle. Table 1 represents a machine state transition when the machine executes the scheduled code. The columns of the sequential state and speculative state indicate registers (e.g. `r2`) and store buffer entries (e.g. `sb1`) where a write, commit, or squashing occurs in each cycle. In particular, the column of a write into the speculative state also indicates a predicate which is written along with data. The column CCR indicates the transition of each branch condition (`c0`, `c1`, and `c2`). All values are initially unspecified.

In the first cycle, instructions `i1` and `i15` are executed. The predicate of instruction `i1` is `alw` (which means "always"), so the execution is non-speculative. Therefore, the instruction writes the loaded data into the sequential state of register `r1`. The actual write occurs at the second cycle because the latency of load instructions is two cycles. In contrast, the execution of instruction `i15` is speculative since both branch conditions of the predicate `c0&c1` are not specified yet. Therefore, the instruction writes the result into the speculative state of register `r2` along with its predicate. In the second cycle, instruction `i10` is executed. It writes the loaded data into the speculative state of register `r5` in the next cycle since the value of its predicate is not specified yet. The execution of instruction `i14` is also speculative. This appends data to the store buffer (`sb1`). In the third cycle, non-speculative instruction `i2` writes the result into register `r3`. Instruction `i16` writes the result into the speculative state of register `r7`.

In the fourth cycle, instruction `i3` defines the branch condition `c0` to true. Notice that the predicate of a condition-set instruction is "alw" regardless of its control dependence because the compiler does not re-allocate an entry of CCR. Predicate `!c0` evaluates to false in the next cycle. As a result, the speculative data in register `r5` is squashed. Also, the execution of instruction `i6` is committed during the execution (notice its latency is two cycles) since its predicate `c0` evaluates to true. In the sixth cycle, instruction `i7` defines branch condition `c1` to true. Thus, predicate `c0&c1` evaluates to true in the next cycle. As a result, the speculative data in registers `r2` and `r7`, and in the store buffer entry `sb1` are committed. Jump instruction `i12` is squashed because its predicate evaluates to false. In the seventh cycle, the control is transferred to the next region (`L8`) by instruction `i17`.

3.5 Handling Speculative Exceptions

Our mechanism of handling speculative exceptions buffers speculative exceptions with a predicate label until the commit point. The buffering is accomplished by setting flag E in the destination entry of the instruction which caused the speculative exception. If the predicate evaluates to true in a later cycle, the outstanding speculative exception is detected. Although this postpones speculative exceptions until the commit point, we need to provide a mechanism to recover the machine state. Our mechanism provides proper and efficient recovery.

If one or more predicates of the buffered speculative exceptions evaluate to true, the detection of speculative exceptions is signaled. The speculative state is then invalidated. This ensures the precise interrupt point. That is, instructions semantically before the excepting instruction are completed, but instructions after the excepting instruction are not. This invalidation of the speculative exception simplifies the recovery of the machine state. The instructions which must be re-executed are simply those speculative instructions which depend upon the commit point. The operands of the re-execution instructions are re-generated by this re-execution. Thus, compiler's scheduling constraints which are due to preserving the availability of the operands are significantly reduced[3].

Since the instructions which must be re-executed are those speculative instructions which depend upon the commit point, they exist only between the top of the current region and the commit point. This is true because the speculative state is closed in the region. To choose instructions to be re-executed and identify the original excepting instruction, two conditions are saved on the commit: the *current condition* and *future condition*. The current condition is a set of branch conditions immediately before the commit point; the future condition is a set of branch conditions at the commit point. At the point right after the exception commit, instructions between the top of the current region and the commit point fall into the following three categories:

1. Instructions whose predicate evaluates to true or false with the current condition must not be re-executed since the instructions of true predicate have already updated the sequential state and the instructions of false predicate should not update any state.

2. Instructions whose predicate evaluates to true or false with the future condition may need to be re-executed. We re-execute all instructions in this category as speculative instructions. Since the value of the predicate is known with the future condition, a caused exception is handled only if the predicate evaluates to true with the future condition; otherwise, the exception is ignored.

3. Instructions whose predicate evaluates to an unspecified value with the future condition are speculatively re-executed as before.

When the speculative exception is committed, we suppress the update of CCR; instead, the new value for CCR is written into a special register called the future CCR. Thus, at this point, CCR holds the current condition and the future CCR holds the future condition. The mechanism then rolls the process back and initiates re-execution. This re-execution will eventually reach the original commit point of the exception and the branch conditions will become those in the future CCR.

The roll-back point is the top of the region. To supply this address in the recovery process, the instruction address is always saved into a special register called a *region program counter* (RPC) by hardware when the control is transferred to a target region from the previously executed region. The machine sets the content in RPC to PC to roll back the process. In the re-execution, if the predicate of an instruction evaluates to true or false in the control path, the instruction is squashed; otherwise (i.e., if the predicate evaluates to an unspecified value) the instruction is executed. Since the original speculative exception is not handled yet, the instruction causes the exception again during the re-execution. If the predicate of the excepting instruction evaluates to true with the future condition, the exception is handled; if the predicate evaluates to false, the exception is squashed; otherwise the exception is buffered like the speculative exception in the normal execution mode.

3. Scheduling is restricted when a non-speculative instruction in a join block attempts to overwrite a register which has been referred by a speculative unsafe instruction. The compiler duplicates the join block to avoid this constraint (if beneficial). Refer to the related discussion in Section 4.2.2.

```
i1:    c0      ?  r1.s = r2
i2:    alw     ?  c0 = r3 < 0
i3:    c0      ?  r2 = load(r2)
i4:    c0&c1   ?  r3.s = load(r4)
i5:    c0&!c1  ?  r5.s = load(r6)
i6:    c0&c1   ?  r7.s = r7 + r3.s
i7:    alw     ?  c1 = r2 > r8
```

Figure 5: Code segment

This execution control slightly modifies the normal execution control. To differentiate this execution from the normal execution, our machine has an execution mode which we call a *recovery mode*. The recovery mode is completed when the process reaches the original commit point of the speculative exception. Since the commit of outstanding speculative exceptions is one cause of exceptions, the exception address is saved in the exception program counter or EPC like other usual exceptions. Thus, the end of the recovery mode is detected by comparing PC with EPC. The recovery mode finishes with copying the future condition to CCR.

We will present a simple example. Figure 5 lists the code segment of a region. Assume that the machine issues a single instruction per cycle. Consider instruction `i2` defines `c0` to true and then instruction `i4` causes an exception. Since `i4` is a speculative instruction, the exception is not handled; instead, flag E of destination register `r3` is set. Similarly, instruction `i5` causes an exception, thus setting flag E of register `r5`. In the seventh cycle, instruction `i7` defines `c1` to true. Since the predicate of register `r3` evaluates to true, the outstanding exception is detected. The mechanism then suppresses the update of CCR and writes the new value for CCR into the future CCR. At this point, CCR holds {`1,U`} and future CCR holds {`1,1`} (each element represents `c0` and `c1`, respectively. `U` represents an unspecified value). The machine switches from the normal mode to recovery mode, and rolls the process back to the top of the region using RPC. Re-execution is then initiated. During the re-execution, instructions `i1`, `i2`, and `i3` are squashed because the predicate of these instructions evaluates to true by referring to CCR (remember only speculative instructions are issued during the recovery mode). Instruction `i4` is re-executed because its predicate evaluates to an unspecified value by referring to CCR. This instruction causes an exception again. Since its predicate evaluates to true by referring to future CCR, this exception is handled this time. Although instruction `i5` is also re-executed and causes an exception, it is not handled because its predicate is evaluated to false by referring to future CCR. Instruction `i6` is then re-executed, and re-generates the speculative state of register `r7`. The recovery mode ends when the process reaches the point `i7`, which is the original speculative exception commit point.

As described in Section 3.4, an instruction word explicitly specifies the state of a source operand (i.e., whether the operand exists in the speculative state or not). Since the operand state is not necessarily preserved between normal execution and re-execution, we need to guarantee the ability to fetch a correct operand in both execution modes. Specifically, an instruction which fetches desired data by specifying the speculative state fetches a wrong operand in the re-execution if the following sequence is taken: that instruction fetches the speculative operand, and completes the execution; later the operand is committed, and some instruction causes a speculative exception; the speculative exception is detected, and re-execution is initiated; that instruction attempts to read the operand in the re-execution, but fetches the wrong operand. We solve this problem through the operand fetch hardware. That is, when an instruction attempts to read a shadow register which holds invalid data, the operand fetch hardware simply reads the sequential register instead of the shadow register. This is correct because the operand was committed before the detected point of the speculative exception, and any succeeding instruction which may overwrite the register of that operand is not committed yet because any instruction which depends upon re-execution instructions are not committed yet. This modification inserts just one gate into the address decoder of the register file. Further description is omitted to conserve space.

Table 2: Benchmark program

Program	Lines	R3000 Cycles	Remarks
compress	1,557	21.3M	Data compression
eqntott	3,441	1,351.6M	Boolean equation minimization
espresso	13,511	1,119.9M	Optimization of PLA structure
grep	430	15.8M	String search
li	7,429	1,245.5M	Lisp interpreter
nroff	7,276	56.0M	Formatting document

4 Evaluation Results

We evaluated the performance in cycle counts. The base machine is the MIPS R3000 [9]. The benchmark programs listed in Table 2 include three SPEC benchmark programs and three UNIX utilities. We used *pixie* to obtain the R3000 cycles. Pixie is a utility that collects dynamic statistics for programs that run on MIPS machines. The statistics include the cycle counts when we run a program on a MIPS machine. The memory system is assumed perfect.

The starting point of the evaluation is the optimized MIPS assembly code. That is, we compile the benchmark programs into assembly code by the MIPS compiler with optimization, and the assembly programs are the source for counting cycles of both R3000 and our machine. For R3000, we compile the assembly code by the MIPS compiler again, and count the cycles by pixie. For our machine, the assembly code is scheduled by our instruction scheduler, and we count cycles using the trace information of the R3000 code by pixie. The speedup over R3000 is calculated by the total number of the R3000 cycles divided by the total number of the evaluated cycles.

If not explicitly specified, our base VLIW machine is assumed to have four ALUs, four branch units, two load units, and one store unit. CCR is assumed to have four entries. Up to four instructions can be issued in parallel. The latencies of the instructions are similar to those of R3000. The latency of branch instructions is assumed to be reduced using a branch target buffer (BTB) [11]. Since the predicated execution eliminates a significant number of branches, the amount of branch penalties is also reduced with given size of BTB. We optimistically assume the branches which are predictable using BTB impose no penalty while other branches such as register indirect jumps impose a one-cycle penalty. This optimistic assumption increases the evaluated performance a few percent according to our cycle-by-cycle simulation. The latency of load instructions is two cycles; the latency of other instructions is one cycle.

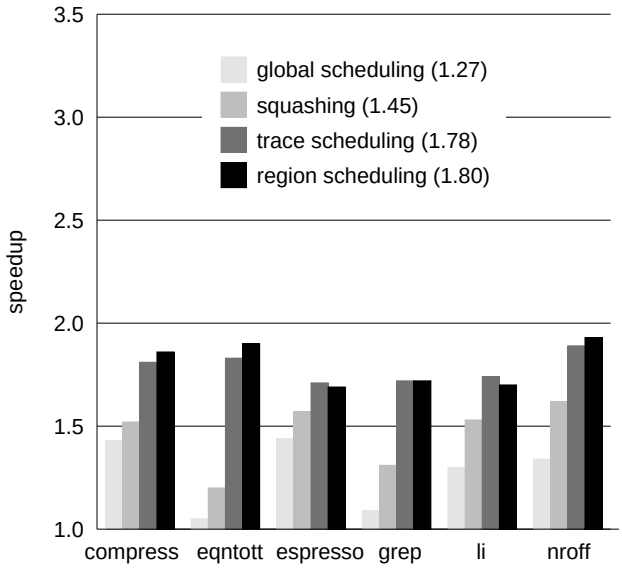

Figure 6: Performance of restricted speculative execution

In this section, we first evaluate restricted speculative execution models as a base experiment. These models need the smallest amount of hardware to support speculative execution. We then evaluate augmented architectures with the predicating mechanism. Two options of implementation are discussed. Since our mechanism needs additional hardware, we also evaluate the amount of the additional hardware.

4.1 Restricted Speculative Execution Models

We evaluate several speculative execution models by incrementally adding hardware supports and scheduling techniques. First, we evaluate machines without the predicated state buffering mechanism. In this machine, the instruction scheduler is limited in speculative code motions. Figure 6 shows speedup for four different speculative execution models.

The *global scheduling model* speculatively moves illegal register instructions across basic block boundaries using register renaming, but does not move illegal memory instructions or unsafe instructions. To eliminate the data dependences upon the replaced copy instruction, *copy propagation* [1] optimization is applied after register renaming. Furthermore, we eliminate the copy instruction if the copied variable is no longer used [1]. In this model, as in percolation scheduling [15], our scheduler iteratively applies several transformations between adjacent basic blocks to increase IPC until no more improvement is found. Delete transformation which deletes a basic block without any instructions except a jump instruction, and node duplication on an instruction movement from a join block are also applied as in percolation scheduling. Loop unrolling and procedure inlining are not applied. Since this model does not need any hardware support for speculative execution, the evaluated performance is one in pure compiler-based approaches. As shown in Figure 6, the speedup over the scalar machine is only 1.27x as a geometric mean.

The *squashing model* supports speculative execution through pipeline squashing. This model allows the compiler to schedule unsafe instructions so that the instruction can be squashed in the pipeline before the write. That is, the pipeline control squashes the side effect of the speculative exception. This squashing model exhibits the performance in squashing speculation stated in Section 2.2. Although this model requires the minimum support for speculative execution, the speedup is limited to 1.45x over the scalar machine, or 1.14x over the global scheduling model.

The *trace scheduling model* picks up a trace in the program and moves instructions within the trace. This scheduling model is superior to our global scheduling model in terms of the window size of scheduling. Unlike the global scheduling which iteratively moves instructions between adjacent basic blocks, the trace scheduling knows the critical paths across several basic blocks. This prior knowledge can put the highest priority on the instructions in the critical path. Furthermore, the branch instructions which fall into the next block in the trace are eliminated. This simply reduces the path length. Our trace scheduler speculatively moves instructions using register renaming and pipeline squashing. The trace does not include any loop back edge since the trace begins with the loop head and ends in the loop tail. Yet, our trace scheduler attempts to move instructions from the trace head to the trace tail by some iterative and backtracking algorithm since code motions along a loop back edge are expected to be beneficial. The achieved speedup is 1.78x over the scalar processor, or 1.40x over the global scheduling model.

The *region scheduling model* allows code motion in a region described in Section 3.3. Since the code motion is not restricted in the trace, this scheduling model uses the larger window of scheduling and is expected to have benefits for branch-unpredictable applications. An instruction is predicated with a branch condition, and the model employs simple predicated execution [13]; if the condition referred to by the predicate is false, the instruction is squashed. Like the previous two models, this model supports squashing speculation. Thus, this model needs the least hardware to support speculative predicated execution. Although the compiler is given more freedom in instruction scheduling than the trace scheduler, the speedup over the trace scheduling model is not significant. Since speculative execution is restricted with limited hardware support, the additional scheduling ability is not beneficial. It is interesting that the additional ability significantly improves the performance if our predicated state buffering mechanism is introduced (described later).

4.2 Speculative Execution Models with Predicating Mechanism

We next evaluate augmented models which employ our predicating mechanism, in which the speculative state is buffered. Two implementation options are proposed and evaluated in this subsection.

4.2.1 Implementation Options

The *region predicating model* fully supports predicating. In this model, unrestricted speculative code motions are allowed. The most complex part in the predicating mechanism is the hardware to evaluate a predicate. As described in Section 3.2, the evaluation of a predicate is a masked match operation between two vectors, the predicate and CCR. The signal delay of this operation never affects the cycle time adversely because it is only a three-gate delay: XOR

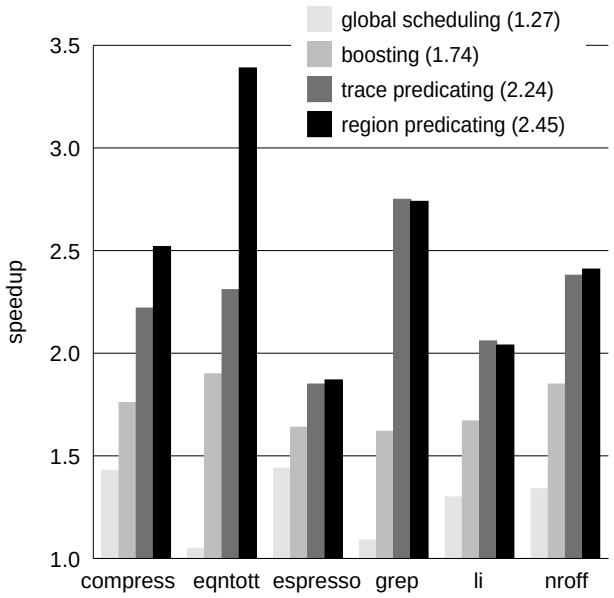

Figure 7: Performance comparison of predicating speculative execution with conventional speculative execution

gate for comparison of each entry + OR gate for masking + AND gate for obtaining total match. The evaluated result of a predicate is used to flip the flags associated with a shadow register. The total match operation and flipping the flags should be completed within a half of the cycle time in our implementation. This signal delay obviously never affects the cycle time[4].

Additional transistors are needed to implement the predicated register file. The additional storages for the speculative state need 76% of the transistors of an 8-read, 4-write, normal register file with 32 registers. Furthermore, the commit hardware including the storages for predicates, predicate evaluation hardware, and a few flags contains 31% of the transistors of a normal register file. As a result, the predicated register file contains 107% more transistors than a normal register file. Although the number of transistors required by the register file for predicating is almost doubled, this hardware increase is acceptable since a register file contains just a few percent of the total transistors in current high-end microprocessors.

Additional bits are required for instruction encoding. The predicate part in an instruction word needs 2xK bits, where K is the number of branch conditions the architecture defines. Furthermore, one bit for each source register is necessary to specify the speculative state. For cost-effective performance, K needs to be three or four (discussed later). Thus, about one byte extension is required for instruction encoding.

The *trace predicating model* is another way to implement predicating. In this model, the compiler's ability is limited in a trace, though the compiler still takes advantage of the hardware support of predicating. This model has benefits in terms of the size of an instruction word; the predicate part needs only $\log_2 K$ bits, since the predicate of an instruction can be encoded with the number of branch conditions the instruction is dependent upon. For this semantics, the compiler may change the condition-set instruction from the original scalar code. Consider the following original scalar code:

```
set c1 if r1 < r2;
branch LAB if c1;
r3 = r4 + r5;
```

If the branch is predicted to be untaken, our predicating scheduler converts the above code as follows:

```
alw ? set c1 if r1 ≥ r2;
c1  ? r3 = r4 + r5;
```

Notice that the condition that sets `c1` is changed for the semantics.

The encoded predicate is converted into the vector form of the region predicating model at run-time. The rest of the execution mechanism is the same as the mechanism of the region predicating model. Our vector-form predicate for buffered speculative results is superior to a counter-type predicate, where a counter is used to represent a predicate. In this representation, the encoded predicate in an instruction word sets an initial value of the counter on a speculative write. The counter is decremented if a condition-set instruction sets a branch condition. In this mechanism, the predicate looses precise control dependence information because the counter cannot specifically represent which branch condition is set. Thus, in this mechanism, the condition-set instructions must be executed sequentially. However, reordering of condition-set instructions is allowed in our vector form representation of a predicate.

4.2.2 Performance Evaluation

Figure 7 compares the speedup for two augmented models described above with two other conventional models, the global scheduling model (previously described) and the *boosting model*. This boosting model is similar to the implementation of boosting described in [18]. The boosting model allows unconstrained speculative code motions within a trace by labeling a speculative instruction and its result with the number of its dependent branches. Basic blocks are maintained from the scalar code. Our boosting scheduler[5] removes any restriction in speculative code motions from our global scheduler if the code motion is along a predicted path. Boosting imposes hardware cost similar to that in predicating since shadow structures are required. As shown in Figure 7, the boosting model achieves 1.74x speedup over the scalar processor, or 1.37x over the global scheduling model through the sufficient hardware support for speculative execution.

The trace predicating model achieves 2.24x speedup over the scalar processor, or 1.76x over the global scheduling model. Compared with the trace scheduling model and boosting model, this performance improvement is significant for both. The trace predicating model has advantages over the trace scheduling model in degrees of speculative execution freedom, and has advantages over boosting in path length reduction through branch elimination and

4. Register file read is another possible critical path, but just a single gate is inserted into the address decoder to select one of two word lines for sequential and shadow registers.

5. The scheduling algorithm described in [18] is more efficient than our boosting scheduler in terms of the bookkeeping ability. We believe that this inefficiency has little impact on the performance in our evaluation where the machine has abundant resources.

Table 3: Prediction accuracy of successive branches

#branches	1	2	3	4	5	6	7	8
compress	.88	.76	.66	.56	.46	.36	.27	.22
eqntott	.87	.77	.68	.61	.56	.53	.50	.49
espresso	.85	.72	.62	.54	.47	.41	.36	.33
grep	.97	.95	.93	.90	.88	.86	.85	.83
li	.88	.77	.68	.61	.55	.49	.43	.38
nroff	.98	.96	.94	.93	.91	.89	.88	.86

reordering of condition-set instructions.

The region predicating model achieves 2.45x speedup over the scalar processor, or 1.93x over the global scheduling mode. The performance improvement over the trace predicating model varies among benchmark programs. This is primarily due to static branch prediction accuracy. The limitation in the trace predicating model causes little impact on performance if the branches are extremely predictable. Table 3 shows the prediction accuracy for successive multiple branches in the benchmark programs [2]. As shown in Table 3, grep and nroff are extremely branch-predictable while other benchmark programs are not. For example, the prediction accuracy for four successive branches in grep is 90%, while the prediction accuracy for four successive branches in compress is only 56%. For grep and nroff, region predicating has no benefit over trace predicating. However, for eqntott and compress, region predicating considerably improves the performance since branches are not extremely predictable. Although branches are not extremely predictable in espresso and li, region predicating shows no improvement over trace predicating. We looked through the code, and found that there seems to be little available ILP to exploit even with our hardware support for speculative execution.

Region predicating slightly lowers the performance under trace predicating in grep and li. This arises from a scheduling constraint caused by dependence which we call *commit dependence*. If it is unknown whether an instruction should refer to a speculative or a non-speculative value, this instruction cannot be scheduled until the speculative value is committed or squashed. In general, if an instruction `i` depends on an instruction `j`, where the block which contains `j` does not dominate the block which contains `i`, `i` must be scheduled after `j` is committed or squashed. In trace predicating, commit dependences do not exist because a single trace ensures that any block is dominated by all preceding blocks in the region, while, in region predicating, they may exist. Our current compiler drives region growth using branch prediction so that only beneficial edges of CFG are added into the region. If the compiler finds that it is beneficial to add only a single entry edge into a join block and not others, the block is duplicated so that the block has only a single entry edge; thus instructions in this block never have commit dependences. Although our current compiler estimates this speculation/dependence trade-off, estimation is inaccurate due to its simple heuristics, and thus region predicating lowers the performance in some cases.

Figure 8 compares the performance improvements for full-issue machines with various amounts of resources. The full-issue machine is a machine with fully duplicated resources such as function units, register ports, and D-cache ports. In Figure 8, three bars for each benchmark program presents the speedup of three different

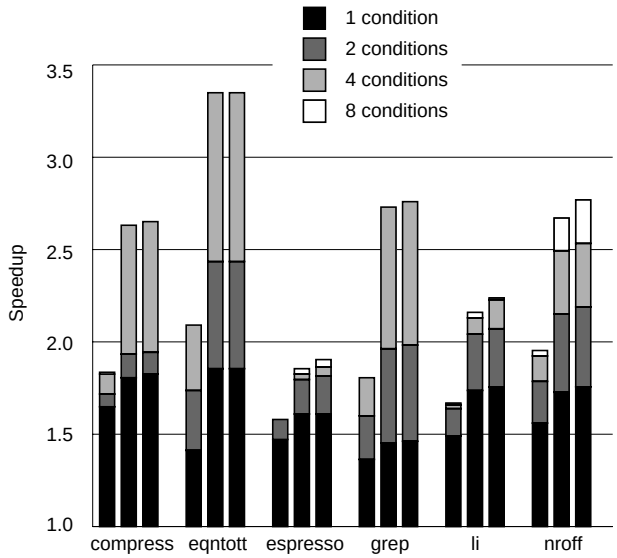

Figure 8: Performance of full-issue machines under various allowable numbers of branch conditions for speculative code motions (left bar: 2-issue, middle bar: 4-issue, right bar: 8-issue)

full-issue machines over the scalar processor: a two-issue machine (left bar), four-issue machine (middle bar), and eight-issue machine (right bar). The lowest portion of each bar represents speedup when the compiler is allowed to speculatively move instructions past one dependent condition. Each of three portions above the lowest portion of each bar represents the increase in speedup when the compiler is allowed to speculatively move instructions past two, four, and eight dependent conditions. As shown in Figure 8, aggressive support for speculative execution is required for a machine with abundant resources. More specifically, the hardware support for speculative execution past two conditions is almost enough to fill issue slots of the two-issue machine, while the hardware support for speculative execution past four conditions is needed to best use the abundant resources of the four-issue machine. Speculative execution past eight conditions or eight duplications of resources, however, produces little impact on performance in our current evaluation. We believe that other compilation techniques which expose more parallelism (e.g. loop unrolling) may be required to exploit more parallelism.

5 Conclusions

In this paper, we have proposed architectural support which provides unconstrained speculative execution. Our mechanism, *predicating*, removes restrictions which limit the compiler's ability to schedule instructions. With predicating hardware support, the compiler is allowed to move instructions past multiple basic block boundaries from any control path. The collapsed block from multiple basic blocks after instruction scheduling is just like a basic block, though it contains instructions with different control dependences. This semantics is that the machine executes instructions in multiple basic blocks simultaneously by simple in-order execution. Our predicated state buffering simplifies the instruction issue and the handling of side effects caused by the speculative execution. Our key idea is that the side effect of speculative exe-

cution is buffered with its predicate, and the buffered predicate efficiently commits or squashes the side effect. Furthermore, we have proposed an efficient mechanism for handling speculative exceptions. The scheme, called the future condition, not only properly postpones the handling of speculative exceptions but also efficiently restarts the process with little penalty on the register and memory system.

We have shown that predicating requires only simple hardware. The predicate which labels the buffered speculative state is evaluated with a simple match operation. Thus, the cycle time is not adversely affected. We have also developed an instruction scheduler to quantify effectiveness of various speculative execution supports for good performance. The evaluation results show that our mechanism significantly improves performance. The four-issue machine achieves 2.45x speedup over the scalar machine, or 1.93x speedup over the global scheduling without speculative execution supports. Our mechanism is an effective mechanism that efficiently supports unconstrained speculative execution, where code motions past multiple basic blocks from any control path are allowed.

Acknowledgment

We thank the anonymous reviewers for their helpful comments. We also especially thank to Monica Lam for her helpful suggestion in our final paper revision. Finally, we thank Yasutaka Horiba and Tadashi Sumi for their encouragement.

References

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

[2] H. Ando, C. Nakanishi, H. Machida, T. Hara, S. Kishida, and M. Nakaya, "Speculative Execution and Reducing Branch Penalty in a Parallel Issue Machine," In *Proc. Int. Conf. on Computer Design*, pp.106-113, October 1993.

[3] R. A. Bringmann, S. A. Mahlke, R. E. Hank, J. G. Gyllenhaal, and W. W. Hwu, "Speculative Execution Exception Recovery using Write-back Suppression," In *Proc. MICRO-26*, pp.214-223, December 1993.

[4] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter, and W. W. Hwu, "IMPACT: An Architectural Framework for Multiple-Instruction-Issue Processors," In *Proc. 18th Int. Symp. on Computer Architecture*, pp.266-275, May 1991.

[5] R. P. Colwel, R. P. Nix, J. J. O'Donnell, D. B. Papworth, and P. K. Rodman, "A VLIW Architecture for a Trace Scheduling Compiler," In *Proc. Second Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp.180-192, April 1987.

[6] K. Ebcioglu and A. Nicolau, "A Global Resource-Constrained Parallelization Technique," In *Proc. Third Int. Conf. on Supercomputing*, pp.154-163, June 1989.

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

[8] P. Y. T. Hsu, and E. S. Davidson, "Highly Concurrent Scalar Processing," In *Proc. 13th Int. Symp. on Computer Architecture*, pp.386-395, June 1986.

[9] G. Kane, *MIPS RISC Architecture*, Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[10] M. S. Lam and R. P. Wilson, "Limits of Control Flow on Parallelism," In *Proc. 19th Int. Symp. on Computer Architecture*, pp.46-57, June 1992.

[11] J. K. F. Lee, A. J. Smith, "Branch Prediction Strategies and Branch Target Buffer Design," *Computer* 17 (1), pp.6-22, January 1984.

[12] S. A. Mahlke, W. Y. Chen, W. W. Hwu, B. R. Rau, and M. S. Schlansker, "Sentinel Scheduling for VLIW and Superscalar Processors," In *Proc. Second Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp.238-247, October 1992.

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

[14] K. Murakami, N. Irie, M. Kuga, and S. Tomita, "SIMP (Single Instruction Stream/Multiple Instruction Pipelining): A Novel High-Speed Single-Processor Architecture," In *Proc. 16th Int. Symp. on Computer Architecture*, pp.78-85, June 1989.

[15] A. Nicolau, "Percolation Scheduling: A Parallel Compilation Technique," Computer Sciences Technical Report 85-678, Cornel University, May 1985.

[16] J. E. Smith and A. R. Pleszkun, "Implementation of Precise Interrupts in Pipelined Processors," In *Proc. 12th Int. Symp. on Computer Architecture*, pp.36-44, June 1985.

[17] M. D. Smith, M. S. Lam, and M. A. Horowitz, "Boosting Beyond Static Scheduling in a Superscalar Processor," In *Proc. 17th Int. Symp. on Computer Architecture*, pp.344-355, May 1990.

[18] M. D. Smith, M. A. Horowitz, and M. S. Lam, "Efficient Superscalar Performance Through Boosting," In *Proc. Fifth Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp.248-259, October 1992.

[19] R. M. Tomasulo, "An efficient Algorithm for Exploiting Multiple Arithmetic Units," *IBM Journal*, 11(1):25-33, January 1967.

[20] D. W. Wall, "Limits of Instruction-Level Parallelism," In *Proc. Fourth Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp.272-282, April 1991.