Register Renaming and Scheduling for Dynamic Execution of Predicated Code

Perry H. Wang Hong Wang Ralph M. Kling Kalpana Ramakrishnan† John P. Shen
Microprocessor Research Labs
Intel Corporation
2200 Mission College Blvd
Santa Clara, CA 95052

Abstract

To achieve higher processor performance requires greater synergy between advanced hardware features and innovative compiler techniques. Recent advancement in compilation techniques for predicated execution has provided significant opportunity in exploiting instruction level parallelism. However, little research has been done on how to efficiently execute predicated code in a dynamic microarchitecture. In this paper, we evaluate hardware optimizations for executing predicated code on a dynamically scheduled microarchitecture. We provide two novel ideas to improve the efficiency of executing predicated code. On a generic Intel Itanium processor pipeline model, we demonstrate that, with some microarchitecture enhancements, a dynamic execution processor can achieve about 16% performance improvement over an equivalent static execution processor.

1. Introduction

In today's modern processor design, one important method of increasing performance is executing multiple instructions per clock cycle. The performance of such processors depends on the amount of instruction level parallelism (ILP) exposed by the compiler and exploited by the microarchitecture. Thus, in the attempt to reach higher performance, cooperation between compiler and microarchitecture has become increasingly important.

Predicated execution is an architectural model where an instruction is guarded by a Boolean operand whose value decides if the instruction is executed or nullified. To explore ILP, a compiler can take full advantage of the predicated execution model by applying a technique called if-conversion. In short, if-conversion [1] is an optimization that converts control flow dependence into data flow dependence. With if-conversion, the compiler can collapse multiple control flow paths and schedule them based only on data dependencies. To achieve enhanced performance with predicated execution, if-conversion requires a detailed analysis [11] on the dynamic program behavior and the dynamic resource availability [3]. On the hardware side, some important issues on predicated execution may involve in its scalability for and compatibility with future-generation machines. Given the availability of increasing transistor budgets, one can expect the incorporation of increasingly more advanced microarchitecture mechanisms. Furthermore, it is important to ensure that a legacy base of predicated code will continue to perform well on future processor generations.

An example of an advanced microarchitecture is that of a dynamic, or out-of-order, execution model, which is in general more complex than a static execution model. Static execution runs code in the order as scheduled statically by the compiler. On the other hand, dynamic execution permits the processor to dynamically adjust instruction scheduling to the run-time behavior of the program. Because of its ability to adapt to the run-time environment, dynamic execution has been employed in many modern processor designs [8][4][7]. The potential performance gains of a dynamic execution engine are facilitated by the following two techniques:

- Register renaming: Registers are renamed to eliminate false dependencies.
- Dynamic scheduling: Instructions are reordered to reduce unnecessary stalls.

There has been a great deal of work on designing dynamic execution processors, but relatively less research on designing those processors with instruction set architecture having full predication capability. Of the research on predication, some have been done on supporting partial predication [13], or conditional moves, found in many dynamic execution processors such as the Alpha processors. Most studies have focused on how predication affects branch pre-

†Kalpana Ramakrishnan is with Sylantro Systems Corporation, 910 E. Hamilton Avenue, Campbell, CA 95008.

0-7695-1019-1/01 $10.00 © 2001 IEEE

diction [14][12][15]. However, there has been little published research on the issues involving register renaming and dynamic scheduling with the presence of predication.

In this paper, we study issues with implementation of dynamic microarchitectures for an instruction set with full predication, and identify potential performance bottlenecks specific to predicated code. To resolve these performance obstacles, we propose microarchitecture solutions that can be realistically integrated into a conventional dynamic execution microarchitecture. The next section discusses in detail the two performance issues on executing predicated code on a dynamic execution machine. Sections 3 and 4 introduce two novel ideas called *select micro-op* (or select-μop for short) and *predicate slip*, which in combination can provide about 16% performance improvement over a comparable static machine, and 7% more than a generic dynamic implementation. Section 5 reports the results of our performance simulations and Section 6 concludes the paper.

2. Two Performance Challenges

To identify potential performance obstacles for a dynamic execution processor in handling predicated code, we first propose a generic dynamic microarchitecture as our baseline performance model. From this model, we observe two performance issues that are induced by executing predicated code on a dynamic execution machine. With careful analysis of the stall conditions in the pipeline, we also identify the root causes behind certain conditions that result in performance loss.

2.1. Baseline Dynamic Microarchitecture

There are several variations of a dynamic execution processor. Typically, the dynamic portion of the processor consists of a register renaming mechanism, which maps between temporary and architectural files, a reorder buffer, reservation stations, and execution units. With all these generic components, the dynamic execution pipeline for our 12-stage integer pipe baseline model is also generic. The dynamic pipeline begins with a 2-stage rename, followed by a register read stage, a 2-stage schedule, an execute stage, and finally a retire stage. In the schedule stage, the instructions first occupy the reorder buffer and then wait in the reservation stations until the data of the source operands become available. After the data are loaded into the register, the instruction enters the execute stage. In the final retire stage, the instructions are retired in order from the reorder buffer.

Conventional dynamic execution microarchitectures employ the use of reservation stations, to remove dispatch blockages due to pending data dependencies in predicate-free code. Likewise for executing predicated code without introducing any special hardware, our baseline model treats the guarding predicate of an instruction as one of the source operands. Thus, our reservation stations check for pending predicate dependency before dispatching the predicated instructions.

2.2. Example

To facilitate illustrating the performance issues and our solutions, we use a code snippet from the *perl* source code in SPECint95. The function is *block_head* in the file *cons.c*. In the middle of this function is an if-and-else statement that picks a variable pointer `arg` and immediately references it.

```
ARG *arg;
if (tail->ucmd.acmd.ac_expr)
    arg = tail->ucmd.acmd.ac_expr;
else
    arg = tail->c_expr;
if (arg) {
    ....
}
```

The control flow graph along with translated pseudo-machine instructions from the C source code is shown Figure 1. The instructions are numbered from `I1` to `I7`, and assume that `r54` contains the pointer `tail`. Instructions `I1` through `I3` in the top block correspond to the *if* conditional test. Instruction `I1` calculates the location of the pointer `tail→ucmd.acmd.ac_expr` and `I2` loads that pointer into `r21`. Instruction `I3` compares the loaded pointer with zero, or null pointer. If the pointer is not null, the program falls through to the left block containing `I5`. Instruction `I5` in this *if* block simply assigns the previously loaded pointer in `r21` to `r44`, a pointer to `arg`. On the right, the *else* block contains instructions `I4` and `I6` that load the pointer `tail→c_expr` and assign it to `r44`. The control flow merges to the bottom block containing instruction I7. This

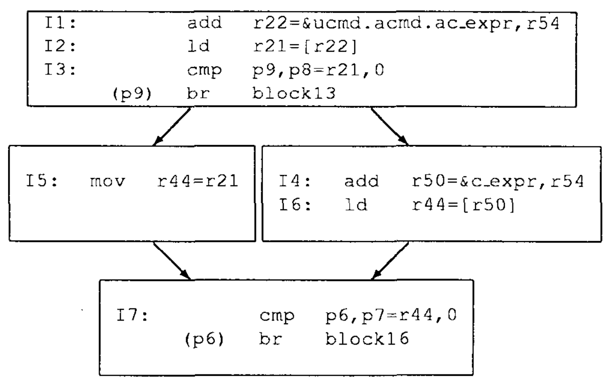

Figure 1. Control flow graph and pseudo code

```
I1:          add   r22=&ucmd.acmd.ac_expr,r54
I2:          ld    r21=[r22]
I3:          cmp   p9,p8=r21,0
I4:   (p9)   add   r50=&c_expr,r54
I5:   (p8)   mov   r44=r21
I6:   (p9)   ld    r44=[r50]
I7:          cmp   p6,p7=r44,0
      (p6)   br    block16
```

Figure 2. After If-conversion

Before Rename

```
I5:   (p8)   mov   r44=r21
I6:   (p9)   ld    r44=[r50]
I7:          cmp   p6,p7=r44,0
```

After Rename

```
I5:   (pQ)   mov   rA=rE
I6:   (pR)   ld    rB=[rF]
I7:          cmp   pS,pT=r(A? or B?),0
```

Figure 3. Renaming issue

instruction reads from `r44` to perform the conditional test on `arg`.

The compiler can recognize this simple hammock as a possible candidate for if-conversion. If-conversion would eliminate the split to the *if* and *else* blocks by predicating and merging the instructions in those blocks. After applying if-conversion, the new straight-line pseudo-machine code is shown in Figure 2. The branch that followed instruction `I3` is removed. In addition, instruction `I5` is predicated by `p8` and instruction `I4` and `I6` by `p9`.

In Sections 2.3 and 2.4 we show that this code, compiled for static execution, can cause unnecessary stalls at the renaming and scheduling stages in our baseline dynamic execution processor.

2.3. Register Renaming

As the instructions enter the renaming stage in our baseline processor, all registers are renamed and each register definition is assigned a unique physical register. Occasionally, the renaming mechanism may need to stall the pipeline if some predicates are not fully resolved. This situation occurs when the renaming unit processes multiple instructions that, guarded by different unresolved predicates, write to a common architectural register. As usual, each definition of this common register would be assigned a unique physical register during renaming. When a consumer instruction that uses the common register reaches the rename stage, the renaming would become ambiguous. Without evaluating the predicates that guarded the definitions of the common register, the renaming unit cannot yet map the common register to the correct physical register. Thus, the processor needs stall the consumer instruction until the predicates are resolved. The stall would induce bubbles in the pipeline and may result in performance loss.

In our example, both the `mov` instruction `I5` and the `ld` instruction `I6` assign their results to the same architectural register `r44`, and they are guarded by `p8` and `p9`, respectively. Figure 3 shows the instructions before and after renaming. To distinguish from the architectural registers, we use alphabetical letters to identify the physical registers. After renaming, the definitions of `r44` by `I5` and `I6` are assigned to `rA` and `rB`, respectively. With unresolved predicates, the renaming of `r44`, the source operand of the consumer instruction `I7`, cannot yet be determined for using either `rA` or `rB`. Thus, until the predicates are resolved, the processor needs to stall `I7` to prevent it from entering the rename stage.

Figure 4 illustrates the example code advancing through the pipeline. We assume that all integer operations take 1 cycle and load instructions 2 cycles. Instruction `I3` defines the predicates used by `I4`, `I5` and `I6`. As instruction `I7` arrives before the rename stage at cycle `n`, it has to stall until `I3` produces the predicate value at cycle `n+4`. At that point, the predicates are resolved and the processor knows which copy of `r44` defined by either `I5` or `I6` can be used. Only then, instruction `I7` can begin at cycle `n+5` to advance through the pipeline and rename its source operand to either `rA` or `rB`. This would result in 3 bubble cycles.

One interesting observation is that any consumer instruction is not required to wait for the resolution of all guarding predicates. It only needs to wait for the youngest defining instruction that is guarded true. Thus, the consumer first waits for the predicate of the last defining instruction to become available. If the predicate turns out true, the consumer can immediately use its defined physical register, despite the outcome of other defining instructions. If the last defining instruction is nullified, the consumer then waits for the predicate of the second-to-last defining instruction. This prioritized checking scheme for the predicate values affects performance depending on the order those values become available.

2.4. Dynamic Scheduling

The other performance limitation occurs during scheduling when the predicated instructions continuously wait in the reservation stations for their predicate-defining instructions to complete. In our baseline dynamic execution model, when a predicate has not yet been produced, all instructions that depend on this predicate need to wait in the reservation stations. Even if all the other source operands are available, the instruction cannot be executed until the predicate is ready.

In situations where some predicates have not been resolved, the reservation stations will start to pile up with those instructions with unresolved guarding predicates. As

Pre-OOO	Rename1	Rename2	RegRead	Reorder	Dispatch	Execute	Writeback	Cycles
I7	I6	I4 I5	I3	I2	I1			n
		I6	I4 I5	I3	I2	I1		n+1
			I6	I4 I5	I3	I2	I1	n+2
				I6	I3 I4 I5	I2		n+3
					I4 I5 I6	I3	I2	n+4
	I7				I6	I4 I5	I3	n+5
		I7				I6	I4 I5	n+6
			I7			I6		n+7
				I7			I6	n+8

Figure 4. Pipeline stall before rename

a result, the reservation stations may get saturated quickly and induce backpressure on the pipeline. This may result in performance loss because of the unresolved predicates at the scheduling stage.

2.5. Problem Analysis

The root cause of the performance limitation in both register renaming and scheduling is from the unresolved predicates. In addition, the renaming issue is triggered by the existence of overlapping variable lifetimes in the program code.

2.5.1. Unresolved predicates. We believe that the major factor contributing to the unavailability of predicate values is the long execution path to the predicates. The predicate-defining instructions are usually on the critical path. From our example, the predicate-defining `I3` depends on the `ld` instruction `I2`. If `I2` triggers a cache miss, the predicate will not be known until after the miss has been serviced. This then lengthens the execution path to the resolution of the predicates. Also, in future processor generations, predicate-defining instructions may take longer to execute as most predicates are produced by compare instructions. Under normal implementation, compare instructions require a serialized propagation of bit-wise operations. Thus, as the clock frequency and the operand size increase, compare instructions may result in multi-cycle latency.

2.5.2. Overlapping variable lifetimes. Through compiler analysis, a variable is deemed live at a point of the control flow graph if its value at that point can reach a subsequent use. The same variable may be defined elsewhere along another control flow path. These paths of multiple variable definitions may meet, resulting in overlapping variable lifetimes. When the compiler picks these paths for an if-converted region, the variable definitions would be assigned a common register, with each variable lifetime guarded by a predicate. The if-and-else statement in our example exhibits overlapping lifetime of variable `arg` in two paths as shown in Figure 1.

2.6. Related Work

There are several solutions to alleviate the stall conditions at the renaming stage for our generic model. A popular way is to transform the predicated instruction to another instruction format similar to the C-style conditional expression, namely *result = (cond) ? expr1 : expr2.* This C-style operation obtains the result from one of the expressions depending on the outcome of the conditional operator. Thus, for the predicated instruction, the physical destination register is assigned from the result of instruction computatoin or from the previously renamed register, depending on the outcome of the predicate. Although this method would avoid stalling the pipeline before the rename stage, it serializes the execution of every predicated instruction due to the dependence on the prior renamed register, thus reducing the effectiveness of dynamic execution. Also, for every predicated instruction, the C-style conditional format would require an extra input operand, which could impose substantial challenge on high-speed circuits.

To remove the extra input load on the C-style conditional move, the Alpha processor decomposes its cmov instruction into two micro-ops [5]. The first micro-op combines the predicate and the previously renamed register into a temporary register. Then the second micro-op selects from either the source operand or the temporary register according to the predicate information embedded in the temporary register. For processors designed with more uses of predicates, this approach may not be feasible due to the creation of excessive micro-ops.

To overcome the performance obstacles, in the following two sections we introduce two new techniques to enhance our baseline microarchitecture. Both techniques solve the

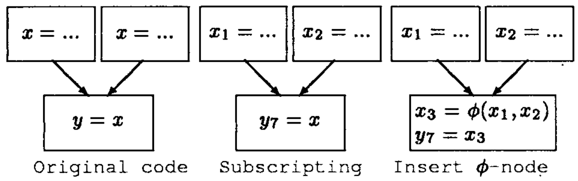

Figure 5. Insertion of phi-node

```
                Before Rename
I5:   (p8)   mov    r44=r21
I6:   (p9)   ld     r44=[r50]
I7:          cmp    p6,p7=r44,0

                After Rename
I5:   (pQ)   mov    rA=rE
I6:   (pR)   ld     rB=[rF]
             select rC = rA, rB
I7:          cmp    pS,pT=rC,0
```

Figure 6. Injection of Select-μop

performance problems elegantly without dynamically modifying the predicated instruction format. The key idea behind these techniques is *not* to implement complex hardware circuitry to solve difficult tasks, but to *postpone* the tasks down the pipeline and solve them with some reasonable change to the existing dynamic execution microarchitecture.

3. Select-μop

To handle the performance issue associating with overlapping variable lifetimes, we propose a novel concept called select-μop that would eliminate the ambiguity of renaming by effectively postponing the renaming task. Without transforming the predicated instructions, this technique would reduce the stall cycles while enable dynamic renaming of registers.

3.1. Rationale behind select-μop

The idea of select-μops is derived from the static single-assignment (SSA) [2][6] representation that is widely used in leading-edge compilers. Single-assignment form guarantees that every target operand is uniquely defined by only one instruction. Thus, when a variable is defined in several basic blocks throughout the control flow graph, each definition instance of the variable is subscripted so that it can be differentiated from other definition instances. Multiple definition instances may reach a common use of the variable, thus it becomes ambiguous as to which of the subscripted variables should be used in the consumer instruction. To solve this problem, the compiler inserts a ϕ-node as a special placeholder at where two definition instances merge. The two subscripted definition variables are used as the source operands of the new ϕ-node, and a new subscripted variable is created as the new destination operand. From that point on, all subsequent uses of the variable are replaced with the new subscripted variable defined by the ϕ-node. The subscripting and inserting of ϕ node is illustrated in Figure 5.

3.2. Select-μop mechanism

Register renaming in a processor model acts similar to subscripting a variable in compiler. As described in Section 2.3, when a common defined register guarded by different predicates is renamed to different physical registers, the consumer instruction cannot rename its source register correctly until the predicates are resolved. In light of the SSA representation, we propose that the processor dynamically introduce special operators named select-μops to defer the exact renaming resolution of physical registers. By injecting it into the instruction stream, the select-μop is used to indicate that multiple renamed registers defined under different predicates may have reached a common use. Those multiple renamed registers and their guarding predicates are assigned to the source operands of the select-μop. A new renamed register allocated for the result of select-μop can then be referenced by all subsequent consumer instructions. As the select-μop later executes, one of its source operands is assigned to the result according to the outcome of the resolved predicates.

With the select-μop mechanism, the consumer instructions do not need to stall for the resolution of the guarding predicates of the defining instructions. At the rename stage, the consumer instructions can safely reference to the destination of the select-μop, knowing that the select-μop will, upon execution, choose the correct value among all the renamed registers. Thus, the renaming ambiguity is delayed and later elegantly deciphered via the execution the select-μops. In essence, the goal of using select-μop is to postpone the resolution of the renaming ambiguity to the latter stages of the pipeline, hence allowing the renaming activity in the early stages to continue.

3.3. Example

We use our example from Section 2.2 to demonstrate the benefit of using select-μops. The register renaming with the injection of select-μops is shown in Figure 6. After both definitions of `r44` are renamed to `rA` and `rB`, the select-μop is injected to select from both of these physical registers and assigns the result to a newly allocated `rC`. After that, instruction `I7` that follows immediately can rename its `r44` to `rC` without stalling the pipeline. Once the select-μop is later executed, it transfers the value from either `rA` or `rB` to `rC` according to the outcome of the resolved predicates. The exact syntax of the select-μop is explained in Section 3.4.2.

Pre-OOO	Rename1	Rename2	RegRead	Reorder	Dispatch	Execute	Writeback	Cycles
I7	I6	I4 I5	I3	I2	I1			n
I7	Sel	I6	I4 I5	I3	I2	I1		n+1
	I7	Sel	I6	I4 I5	I3	I2	I1	n+2
		I7	Sel	I6	I3 I4 I5	I2		n+3
			I7	Sel	I4 I5 I6	I3	I2	n+4
				I7	Sel I6	I4 I5	I3	n+5
					Sel I7	I6	I4 I5	n+6
					Sel I7	I6		n+7
					I7	Sel	I6	n+8

Figure 7. Pipeline with injection of select-μop

Figure 7 shows the pipeline diagram with the use of select-μop. When the condition to stall `I7` is detected at cycle `n`, the select is subsequently injected at cycle `n+1` so that `I7` can immediately follow. Comparing with the baseline scheme, the select-μop eliminates the three bubble cycles. Although select-μops introduce an extra layer of dependency to move the value from one selected register to the other, we save more cycles with efficient dynamic execution.

3.4. Select-μop Microarchitecture

We separate the microarchitecture support for select-μops into two components, one dealing with generating the select-μops, and the other one with executing the select-μops. In this paper, we propose to support select-μop with four source operands, *s0*, *s1*, *s2*, and *s3*.

3.4.1. Register Alias Table with predicates. To support the generations of select-μops, we propose to augment the register alias table (RAT) with predicates. The register alias table is used by the renaming unit to map from architectural register identifiers to physical register identifiers. When an in-flight instruction enters rename, the RAT looks up the physical identifiers of the source operands as well as assigns the result operand with a new physical identifier.

In the augmented RAT, each entry is expanded to have multiple slots, with each slot recording the identifiers of the physical register as well as the guarding predicate of the instruction that defines this physical register. A logic view of the augmented RAT is shown in Figure 8. The number of architectural registers determines the number of rows in the RAT. Each row (entry) is assigned an architectural register whose identifier is used to index to the entry. To support the select-μops with four source operands, each row of this table consists of a valid bit and four slots.

In the rename stage, the augmented RAT operates in three steps for the result register of an in-flight instruction.

- Index into the RAT with the architectural identifier of the result register.
- For the located entry, check the predicate of the in-flight instruction.
  - If the instruction is not predicated, clear the entire entry.
  - If the predicate matches one of the predicates in the slots, clear its associated slot.
- Allocate a new physical register and append to a slot the physical identifier along with the identifier of the guarding predicate.

A select-μop is needed only when two or more slots are occupied. However, to avoid injecting excessive select-μops, select-μops are injected only when they are needed. This demand-driven policy for injecting select-μops is when there are more than one occupied slot in the entry, plus when

- The use of the register is encountered at the rename stage, or
- All slots in the entry are occupied and a new physical identifier is being allocated, or

		slot 0		slot 1		slot 2		slot 3	
	v	PhyReg	Pred	PhyReg	Pred	PhyReg	Pred	PhyReg	Pred
0	1	r4	p1	r8	p9				
1	1	r11	p0						
2	0								
3	1	r6	p9	r14	p4	r21	p0	r17	p2
4	1	r24	p2						
..									
..									
..									
n	1	r7	p12	r15	p3				

Figure 8. Augmented Register Alias Table

- One of the guarding predicates in the slots is redefined.

When any of the above conditions is met, a select-μop is generated. Physical identifiers in all of the occupied slots are copied to the source operands of the select-μop. A new physical register is allocated for the destination operand. The entire entry in the RAT is then cleared and replaced with the new physical register identifier. Once the select-μop is formed, the processor inserts it into the instruction stream.

3.4.2. Select-μop execution. The select-μop enters into the reservation station like any other instructions. Before execution, any of the four source operands is a candidate that holds the value for the destination operand. Except for *s0*, each of the source operands is associated with two status bits, the v-bit and the p-bit. These status bits are used to control the selection of the source operands. The first one of the status bits, the v-bit, is to specify whether the register is ready. The second one, the p-bit, is to indicate whether the renamed definition register has been architecturally committed. When the other source operands are nullified, the select-μop chooses *s0*, the default physical register, which would always be committed. Thus, *s0* is not associated with a p-bit, but only with a v-bit. This format handles at most three parallel predicated instructions writing to the same register.

The priority information of the source operands is also inherent in the select-μop, with *s3* representing the highest priority. When the status bits of *s3* indicate the operand is valid and ready, the select-μop can immediately be executed without waiting on the resolution of the rest of the source operands. The priority of the source operands is laid out, from left to right, in the program order that the instructions are fetched. Thus, the youngest defining instruction always has the highest priority.

In our dynamic execution model, the reservation station receives the results of execution through the bypass network. For the select-μop, the reservation station obtains two broadcasted signals for each of its source operands. One wire is to signal that the computation of the operand has completed and the bypassed data is ready. This is hooked up to the v-bit of the source operand. The other wire indicates that the bypassed data is to be committed or discarded. This goes to the p-bit of the source operand.

The logic that realizes the dispatching condition with the source fan-in of 4 is depicted in the middle of Figure 9. When the highest priority operand (*s3*) is available, *v3* becomes 1. Depending on *p3*, which is the predicate value, the select-μop can be immediately dispatched if *p3* is 1. If *p3* is 0, the select-μop needs to wait for its lower priority operands to become available.

Once the select-μop is dispatched for execution, the value from one of the source operands is transferred to the destination register. The logic that executes the select-μop with source fan-in of 4 is depicted in the lower portion of Figure 9. This logic is simply a cascade of three 2x1 multiplexers. The p-bit is used to toggle the multiplexer select. One should note that this is a logical view of the select-μop execution. The actual circuitry can be implemented in different ways. When a p-bit is set to 1, the output obtains the data from the corresponding source operand. Conversely when a p-bit is set to 0, the data is fetched from the output of another cascaded multiplexer. This logic correctly realizes the priority specified in the select-μop. Once the execution of select-μop completes, the reservation station then receives the result broadcasted for all its uses.

4. Predicate Slip

To deal with performance issue in scheduling, we propose an idea to free up the reservation station entries as soon as instructions are ready for execution, regardless of the value of the guarding predicate. This is called *predicate slip*.

4.1. Rationale behind predicate slip

In the conventional computation model, there are three types of data dependencies: RAW, WAR, and WAW. Dynamic scheduling tries to reduce the stalls of the true or RAW dependencies, while register renaming eliminates the false dependencies, WAR and WAW. Predication is used to transform a control dependency to a (true) data dependency, that is, the dependency between the write of predicate registers and the use of predicate registers for guarding instructions. However, predicate dependency is not truly a RAW dependency; predicate dependency has a special property: *Different from the roles of other source operands, the predicate does not contribute to the result of the instruction computation. However, it dictates whether the result is to be committed.*

This property implies that the predicated instruction can always be executed regardless of the value of the predicate since it does not alter the outcome of the computation. Thus, the guarding predicate can be independently referenced at a different stage in the pipeline. Predicate slip provides performance improvement by taking full advantage of this special property on predicate dependency.

4.2. Predicate slip mechanism

The idea of predicate slip is to postpone the checking of the predicate values until just before the predicated instructions are retired. That is, the predicated instructions are dispatched from the reservation stations as soon as their true (non-predicate) dependencies are satisfied. Thus, without checking for predicate dependency, we continu-

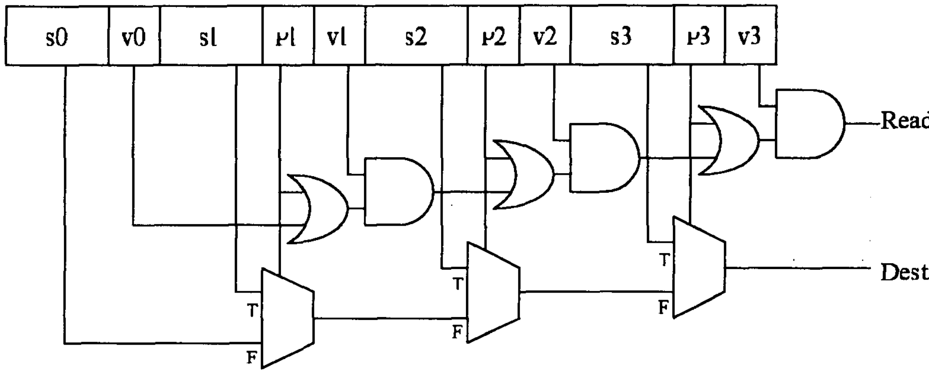

Figure 9. Dispatch and Execution of select-μop

ously dispatch the predicated instructions like any other unpredicated instructions. This is equivalent of predicting a true predicate for all the predicated instructions by assuming that their results will be committed.

We delay the checking of the predicate values until after the instructions are executed. This way, the instructions will not needlessly occupy the entries in the reservation stations. Once they *slip* through execution, the predicated instructions are temporarily held in the reorder buffer. The results of the slipped instructions cannot be written back to the register files, nor can they be bypassed for the consumer instructions. A slipped instruction waits in the reorder buffer until its guarding predicate eventually becomes available. If the predicate is evaluated true, the instruction can be architecturally committed and retired. Also, the result is broadcasted through the bypass network. Otherwise, the result is discarded.

Predicate slip satisfies the property on predicate dependency since it guarantees that the predicated instructions can be safely slipped without altering the result of execution. By slipping the instruction ahead, predicate slip not only de-allocates reservation station entries ahead of time but also helps shorten the execution path. One should also note that select-μops are not considered as candidates for predicate slip, since the predicates used in select-μops directly contribute to the outcome of the computation.

4.3. Example

We again use our example to demonstrate how predicate slip benefits dynamic execution. Using predicate slip mechanism, our reservation stations can now dispatch any instruction when all except predicate dependences are satisfied. Figure 10 shows the latter portion of our dynamic pipeline. Instructions `I4` and `I5` arrive to the reservation stations at cycle `n+3`, and they depend on the predicate generated by `I3`. With predicate slip, this predicate dependence is ignored and all three instructions `I3`, `I4`, and `I5` can be dispatched from the reservation stations simultaneously. At the end of the execution, the predicate generated by `I3` is resolved and broadcasted. At cycle `n+5`, the processor determines whether `I4` or `I5` is to be committed. Because of its ability to slip the instruction through, the processor is able to shorten the execution path and save one cycle in this example.

4.4. Predicate Slip Microarchitecture

To support predicate slip, only minor changes are needed on the existing microarchitecture.

First, the logic that checks for predicate dependency in the reservation station is removed. A check bit and a predicate identifier are then attached to each entry in the reorder buffer. After the predicated instruction slips through execution, the check bit is set and the predicate identifier is recorded. In conventional implementation, the reorder

Reorder	Dispatch	Execute	Writeback	Cycles
I2	I1			n
I3	I2	I1		n+1
I4 I5	I3	I2	I1	n+2
I6	I3 I4 I5	I2		n+3
Sel	I6	I3 I4 I5	I2	n+4
I7	Sel	I6	I3 I4 I5	n+5
	Sel I7	I6		n+6
	I7	Sel	I6	n+7
		I7	Sel	n+8

Figure 10. Pipeline with predicate slip

buffer is a circular queue that advances its pointer after an instruction retires from the buffer. When the check bit of an entry is set, the circular pointer needs to stop advancing and look up the value of the guarding predicate. The pointer continues to hold until the predicate value is fetched. Thus, extra wires are needed for fetching the predicate values from the predicate register file to the reorder buffer. At that time, if the predicate is true, the instruction is retired. If it is false, the reorder buffer entry is clear.

5. Performance Evaluations

This section presents performance evaluation of two microarchitectural enhancements: select-μop and predicate slip.

5.1. Performance Models

For performance evaluation, we use Intel's Itanium simulation environment including the tools and workload. The Itanium architecture [9][10] supports advanced architectural features such as speculation and predication. These architectural features enable compilers to extract more instruction level parallelism by using such special optimizations as if-conversion.

Our simulation environment includes a static (in-order) execution research model and a baseline dynamic execution research model. To maintain a reasonable comparison between the two models, we simply configure the dynamic pipeline model with four more stages (2 for renaming and 2 for scheduling) than the static pipeline model. For the four extra stages in the dynamic pipeline, two stages are accounted for the register rename and the other two for the dynamic scheduling. In addition to the baseline dynamic model, we build a second dynamic pipeline which implements the proposed microarchitectural enhancements.

Using the same benchmark binaries, comparison of results between the two dynamic pipeline models can demonstrate the headroom to improve dynamic pipeline performance. Also, comparison of results between static pipeline model and dynamic pipeline model will reveal the benefits of those two different microarchitectures. The performance gains achieved from dynamic microarchitectures may also shed some light on new compiler optimizations to more effectively utilize dynamic techniques.

5.2. Results

The benchmarks evaluated in our experiment include the entire SPECint95 suite except gcc. An Intel research compiler is used to compile the benchmarks, targeted for the generic static execution model. Figure 11 shows the percent improvement of dynamic execution over static execution. For each benchmark, there are four dynamic execution runs measured against the static execution run. The first run is with our baseline dynamic model. The second one is the dynamic execution run with predicate slip, and the third one is with select-μop. The last run is with both techniques applied. The results of these runs are compared with the static execution run.

The result indicates that our dynamic baseline model has on average 9.1% performance increase over the static model. Note that this performance result should be taken in perspective. On one hand, future advancement in compiler technology should bring the performance of static execution closer to the dynamic execution. On the other hand, further improvement in dynamic microarchitecture may raise the performance of dynamic execution even higher. Hence, the exact performance gap between the static model and the baseline dynamic model reported in this paper may differ for future binaries. In general, these two approaches for improving performance are complementary.

On average, both of our novel microarchitecture techniques give additional performance improvements over the baseline dynamic model. Predicate slip yields roughly 1.2% more and select-μop about 6.4%. The combined effect is at least 7.7%, which is slightly more than the sum benefits with both techniques separately applied. We believe this observation is attributable to the select-μop enabling more predicated instructions to pass through the rename stage and enter the reservation stations. As a result, it allows more opportunity for the predicate slip to slip more instructions through the execution units.

The result shows that the baseline dynamic model does not always outperform the static model. Two benchmarks, m88ksim and go, have the results of baseline dynamic runs lower by 15% and 3%, respectively. For m88ksim, a processor simulation program, the compiler can schedule the code well due to its regular data structure and access patterns. Thus, the static execution processor can efficiently run those highly optimized code, which leaves little opportunity for the dynamic execution to further improve the performance via dynamic scheduling. As the result, the dynamic execution processor is relatively slower because of its added hardware complexity. But with select-μop technique added to the dynamic model, the performance of m88ksim becomes better than that of the static model by about 8%. Similarly for go, the dynamic execution with both of our techniques can still outperform the static model by roughly 1%.

The baseline dynamic runs of ijpeg and li show a moderate 7% improvement over the static run. Vortex improves about 14% and compress 19%. By applying both predicate slip and select-μop, the improvement of compress reaches close to 30%. In particular, for perl, both techniques can boost the dynamic execution performance up to 40% over the static model. Comparison of both techniques indicates that select-μop can reduce more stall cycles than predicate

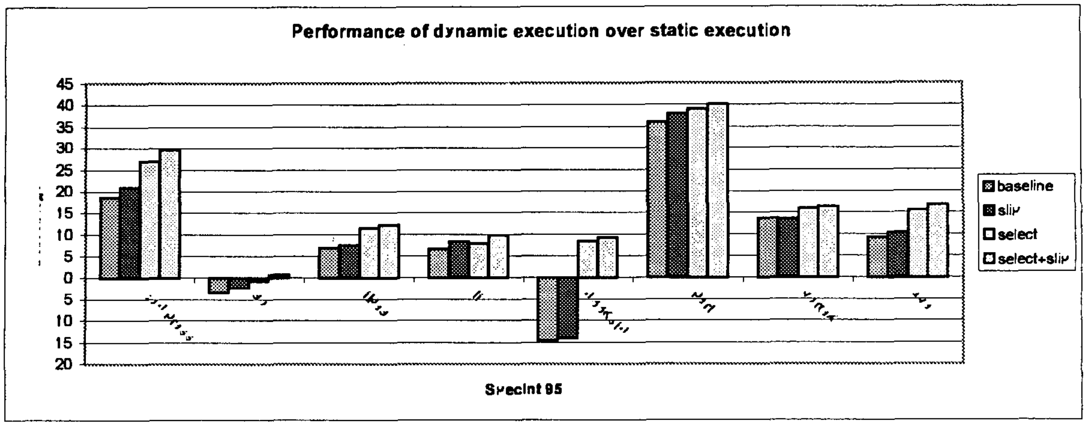

Figure 11. Performance improvement of dynamic execution

slip in most of the benchmarks. Only in li, predicate slip is more beneficial. Overall, the result confirms that our techniques can further exploit the benefit of dynamic execution by removing the unnecessary stalls.

5.2.1. Implementation overhead from select-μop generation. Unlike predicate slip, which is easier to implement, the microarchitecture for select-μops can be implemented with different approaches. To quantify performance tradeoffs, we focus on modeling different implementations for generating select-μops by assuming different number of cycles required for rename. The augmented version of register alias table can potentially require more rename stages than the default two stages that were previously assumed in the baseline dynamic model. Thus, the effect of using multi-stages for rename is modeled and the result is depicted in Figure 12.

Figure 12 shows, with select-μop applied, the percent improvement of dynamic execution runs with and without predicate slip over the static execution run. For both dynamic runs, the rename is extended from the default two stages to as many as five stages. A five-stage rename may be unnecessarily deep, but it serves to demonstrate the effect of a pessimistic implementation. For the combined techniques, the performance improvement on average drops a bit more than one percent for every stage added to the rename. Extending from two to five-stage rename, the performance drops close to 5%.

5.2.2. Overhead from select-μop execution. The impact of executing select-μops in different latencies is modeled. Because of its semantically inherent priority of evaluation, the priority multiplexer logic for executing select-μop in the worst case can be implemented via cascading. Cascaded logic may potentially require more cycles to execute.

Figure 13 shows the percent improvement of dynamic runs with select-μop over the static runs. For each benchmark, the latency of select-μop is varied from 1, 2, 3, 5, to 7 cycles. The result indicates that the execution latency of select-μops does not affect the performance as much as the injection of select-μops at rename. As the execution latency of select-μops is varied from 1, 2, to 3 cycles, performance barely changes. Even when the latency reaches 7 cycles, the average performance drop is still less than 3%. We believe the reason is threefold. First, in a dynamic execution model, the long latency of the select-μops is overlapped with the execution of other independent instructions. Second, most of the select-μops may not be on the critical paths of the execution. And finally, our select-μop injection mechanism is demand-driven, so that the number of select-μops dynamically introduced is relatively small.

6. Concluding Remarks

In this paper, we investigate the important issues regarding efficient execution of predicated code on future dynamic processors. We first identify two potential sources of pipeline stalls on the critical path directly attributed to predicate handling in a baseline dynamic processor. Then we propose two microarchitectural techniques, namely predicate slip and select-μops to overcome these stall problems and enable more efficient execution of the predicated code. The microarchitecture described in this paper can be realized without increasing much complexity in hardware. Using SPECint95 compiled for the Itanium instruction set and the research processor models for both static and dynamic pipeline, we have conducted evaluations of performance tradeoffs for different processor configurations. We have found that predicate slip can provide moderate performance improvements; our result shows an average improvement of 1%. For select-μops, the performance results unveil that the 6% benefit gained even exceeds the potential penalty of ex-

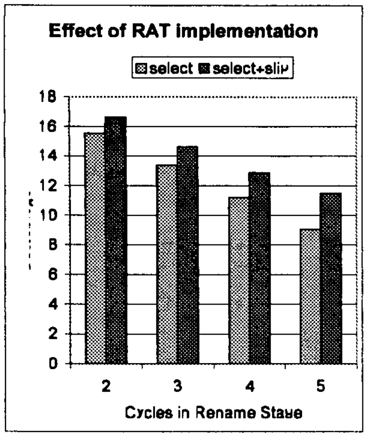

Figure 12. Multi-cycle Select-μop Injection

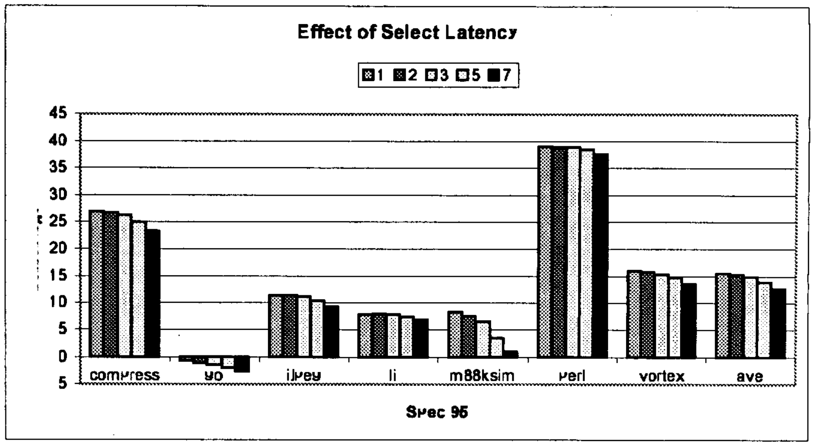

Figure 13. Multi-cycle Select-μop Resolution

ecuting long latency select-μops. Both microarchitectural techniques in combination can bring about additional 7.7% performance boost on top of the 9.1% gain from the baseline dynamic execution. Overall a dynamic execution machine with the proposed optimizations can achieve a total of 16% performance improvement over the static execution machine.

Acknowledgment

The arthors would like to thank Andy Glew, Jared Stark, Bob Rychlik, Jamison Collins, and the anonymous reviewers for their comments.

References

[1] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *Proceedings of the 10th ACM Symposium on Principles of Programming Languages*, January 1983.

[2] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of values in programs. In *Conference Recordings of the 15th ACM Symposium on Principles of Programming Languages*, January 1988.

[3] D. I. August, W. W. Hwu, and S. A. Mahlke. A framework for balancing control flow and predication. In *Proceedings of the 30th International Symposium on Microarchitecture*, December 1997.

[4] D. P. Bhandarkar. *Alpha Implementations and Architecture.* Digital Press, Newton, MA, 1996.

[5] *Alpha 21264 Microprocessor Hardware Reference Manual.* Compaq Computer Corporation, 1999.

[6] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. Technical Report RC14756, IBM, revised March 1991.

[7] J. Heinrich. *MIPS R10000 Microprocessor User's Manual.* MIPS Technologies Inc, September 1996.

[8] *Intel Architecture Optimization Reference Manual.* Intel Corporation, February 1999.

[9] *IA-64 Architecture Software Developer's Manual.* Intel Corporation, 2000.

[10] *Itanium Processor Microarchitecture Reference for Software Optimization.* Intel Corporation, March 2000.

[11] R. Johnson and M. Schlansker. Analysis techniques for predicated code. In *Proceedings of the 29th International Symposium on Microarchitecture*, December 1996.

[12] S. A. Mahlke, R. E. Hank, R. A. Bringmann, J. C. Gyllenhaal, D. M. Gallagher, and W. W. Hwu. Characterizing the impact of predicated execution on branch prediction. In *Proceedings of the 27th International Symposium on Microarchitecture*, December 1994.

[13] 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 22nd International Symposium on Computer Architecture*, June 1995.

[14] D. N. Pnevmatikatos and G. S. Sohi. Guarded execution and branch prediction in dynamic ILP processors. In *Proceedings of the 21st International Symposium on Computer Architecture*, April 1994.

[15] G. S. Tyson. The effects of predicated execution on branch prediction. In *Proceedings of the 27th International Symposium on Microarchitecture*, December 1994.