Recovery Code Generation for General Speculative Optimizations

JIN LIN, WEI-CHUNG HSU, and PEN-CHUNG YEW
University of Minnesota
and
ROY DZ-CHING JU and TIN-FOOK NGAI
Intel Corporation

A general framework that integrates both control and data speculation using alias profiling and/or compiler heuristic rules has shown to improve CPU2000 performance on Itanium systems. However, speculative optimizations require check instructions and recovery code to ensure correct execution when speculation fails at runtime. How to generate check instructions and their associated recovery code efficiently and effectively is an issue yet to be well studied. It is also, very important that the recovery code generated in the earlier phases integrate gracefully in the later optimization phases. At the very least, it should not hinder later optimizations, thus, ensuring overall performance improvement. This paper proposes a framework that uses an *if-block* structure to facilitate check instructions and recovery code generation for general speculative optimizations. It allows speculative instructions and their recovery code generated in the early compiler optimization phases to be integrated effectively with the subsequent optimization phases. It also allows *multilevel speculation* for multilevel pointers and multilevel expression trees to be handled with no additional complexity. The proposed recovery code generation framework has been implemented and evaluated in the Open Research Compiler (ORC).

Categories and Subject Descriptors: D3.4 [**Programming Languages**]: Processors—*compiler; optimization*

General Terms: Algorithms, Performance, Design, Experimentation

Additional Key Words and Phrases: Recovery code, multi-level data speculation, speculative SSA form

1. INTRODUCTION

Control and data speculation [Heggy et al. 1990; Bringmann et al. 1993; Mahlke et al. 1993; Wu et al. 1994; Postiff et al. 2000; Ju et al. 2000; Mahadevan et al. 2000; Lin et al. 2003, 2004] has been used effectively to improve program performance. Ju et al. [2000] proposed a unified framework to exploit both control

Authors' addresses: Jin Lin, Wei-Chung Hsu, and Pen-Chung Yew, University of Minnesota, MN; Roy Dz-Ching Ju and Tin-Fook Ngai, Intel Corp., Santa, CA.

and data speculation targeting specifically for *memory latency hiding*. In their scheme, data speculation is exploited by hoisting *load* instructions across potentially aliasing *store* instructions, thus more effectively hiding the memory latency. In Lin et al. [2003], a framework is proposed to support more general speculative optimizations, such as *speculative register promotion* and *speculative partial redundancy elimination*, in addition to data speculative code scheduling. However, the crucial issues of check and recovery code generation have so far been mostly overlooked.

The recovery code generation scheme used by Ju et al. [2000] (referred to as the JNMW algorithm in this paper) during the code scheduling phase can be understood as follows. When a *load* instruction is speculatively moved up to an earlier program point, a special check instruction, *chk*, is generated at its original location. Some of the instructions that are *flow dependent* on the *load* instruction could also be moved up with the speculative *load* instruction. To facilitate recovery code generation, the compiler introduces additional dependence edges for the new *chk* instruction. This simple scheme works reasonably well because the generated recovery code rarely interacts with other compiler optimizations (*instruction scheduling* is usually performed near the end of compiler optimization phases). In Lin et al. [2003], a check instruction is explicitly modeled as an *assignment statement* and the corresponding recovery code is generated separately. These additional assignment statements could hinder the subsequent optimizations and impact the performance of the speculative optimizations, unless we take extra care of them in later analyses and optimizations. This would make later analyses and optimizations more complex and less effective.

In this paper, we propose a unified framework for recovery code generation that supports general speculative optimizations including speculative partial redundancy elimination (PRE) (usually performed early) and instruction scheduling (usually performed late). It uses an explicit control flow structure to model check instructions and recovery code. Using this model, the check instruction and its recovery code can be treated as a highly biased *if-block* in later analyses and optimization phases. The check instructions and their recovery blocks can thus be integrated seamlessly in the later phases such that they are analyzed and optimized as if they were ordinary instructions without special treatment [Ju et al. 2000; Lin et al. 2003]. This approach is most obvious for handling *multilevel speculation,* such as *cascaded speculation* for multilevel pointers [Ju et al. 2000]. To the best of our knowledge, this is the first general compiler framework for recovery code generation. Our experimental results show that the proposed framework can effectively support general speculative optimizations.

Compared to our proposed *if-block*-based recovery code generation, existing speculation approaches either delay data speculation until final instruction scheduling or greatly constrain effective code motion applicable to early speculative optimizations [Ju et al. 2000; Mahlke et al. 1993; Lin et al. 2003]. Our work advances the field by providing a structured and general approach to model speculation and its associated recovery code while not limiting optimization opportunities. It transforms the problem of the speculation of dependent instructions into partial ready code motion that a good instruction level

parallelism (ILP) compiler already tackles. Furthermore, the control flow generated for the speculation check comes with its inherent control profile (i.e., the branch to the recovery block is rarely taken) in our presented model.

The rest of the paper is organized as follows. Section 2 describes our scheme to model check instructions and their recovery code generation using explicit *if-block* structures. In Section 3, we show how to generate recovery code for both single and multilevel speculation in our speculative PRE framework. Section 4 discusses how to model the speculation of dependent instructions as partial ready code motion. In Section 5, we discuss how to eliminate redundant check instructions. Section 6 presents and discusses experimental results. In Section 7, we review previous recovery code generation algorithms and explain why they are inadequate for handling general speculative optimizations. We give our conclusions in Section 8.

2. A FRAMEWORK FOR CHECK INSTRUCTIONS AND RECOVERY CODE GENERATION

A proper representation of a *check instruction* and its corresponding *recovery block* is essential in supporting speculative optimizations. A *recovery block* is the recovery code for a specific *check* instruction. In the rest of the paper, we will use *recovery block* and *recovery code* interchangeably. In a compiler, intermediate representation (IR) is important, because they could interact with all optimization phases in a very complicated way. In this section, we present our model and IR for check instructions and recovery code.

We *explicitly* represent the semantics of the *check* instruction and its corresponding *recovery block* as a conditional *if-block*. The *if-block* checks whether the speculation is successful or not. If not, the recovery block will be executed. Initially, only the original load instruction associated with the *speculative load* is included in the recovery block. For both data and control speculative optimizations, checks and recovery code are represented as *if-blocks*. For data speculation, the *if-block* can be inserted either at the original location of the speculative load or after the *last weak update*, but *before* the use of speculative load. The term *weak update* means the update is unlikely to be aliased with the load and thus can be speculatively ignored [Lin et al. 2003]. For speculative code scheduling [Ju et al. 2000], the speculative load refers to the load which is determined to be speculatively hoisted across the potentially aliased store. For speculative PRE [Lin et al. 2003], the speculative load refers to the load, that is marked with speculative redundant flag. For control speculation, the *if-block* can be inserted at the original location of the speculative load. Since the compiler selects data speculation that has a low aliasing probability, mis-speculations should be unlikely to happen. Similarly, checks for the selected control speculation should also be unlikely to fail. Hence, the compiler marks the *if-condition* as *highly unlikely to be true*. Based on such an *explicit* representation, later analyses and optimizations can treat this *if-block* as any other*highly biased if-blocks.* There will be no need to distinguish *speculative* code from *non-speculative* code during later analyses and optimizations, as will be discussed in Section 4.

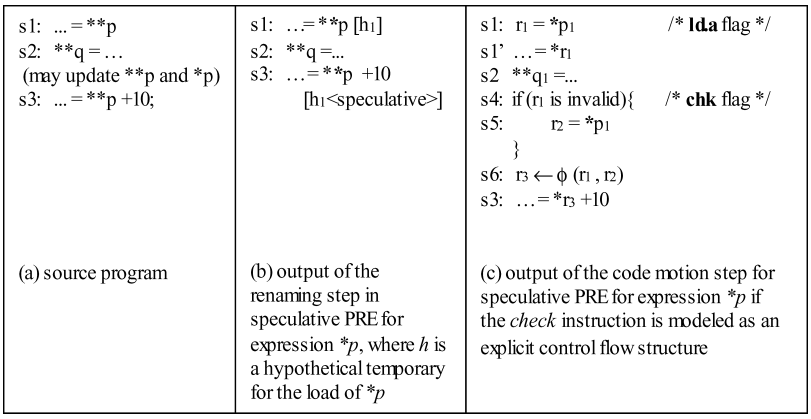

Fig. 1. Example of recovery code generation for speculative PRE.

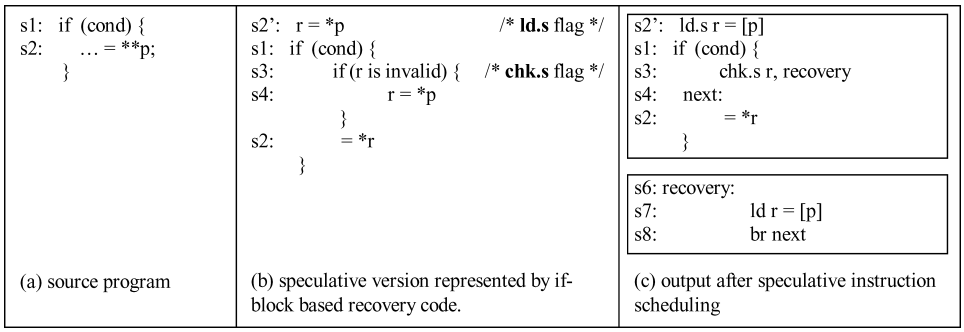

Fig. 2. Example of recovery code generation in speculative instruction scheduling.

We use one example code in Figure 1a to illustrate the effect of our proposed approach for speculative PRE [Lin et al. 2003]. We assume that $^{**}q$ in s2 could be potentially aliased with the loads $^{*}p$ and $^{**}p$ with a very low probability. The second occurrence of the expression $^{*}p$ in s3 has been determined to be speculatively redundant to the first one in s1, as shown in Figure 1b. The compiler inserts an *if-then* statement (in s4) after the weak update $^{**}q$ in s2 for the speculative redundant expression $^{*}p$, as shown in Figure 1c. The *then* part of the *if-then* statement initially has just one single *load* instruction (i.e. the original *load* instruction). The *chk* flag is used to mark those *if-blocks* that correspond to check instructions. Those *if-blocks* will be converted to *chk.a* instructions, or *ld.c* instructions (if the load is the only instruction in the recovery block) during code generation.

Similarly, this recovery representation can also be applied to the control speculation without any change. Considering the program in Figure 2a, the frequency/probability of the execution paths can be collected by the edge/path profiling at runtime and represented in the control flow graph. If the branch-taken path (i.e., the condition being *true*) has a high probability, the compiler can move the load instruction across the branch and execute it speculatively using the *ld.s* instruction. A check instruction (*chk.s*) is inserted at its home

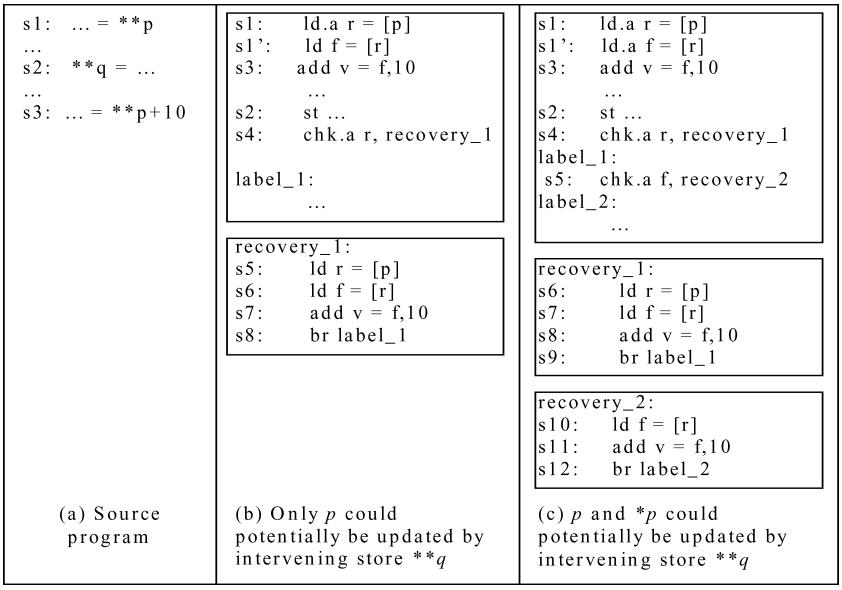

Fig. 3. Examples of multilevel speculation.

location to catch and recover from any possible exception. The *ld.s* and *chk.s* are Itanium in structions that support control speculation. Figure 2b shows how we use *if* statement to represent the recovery code. The later code generation phase converts this special *if* statement into *chk.s* instruction, the load instruction in s2′ into *ld.s* instruction, and generates the recovery block as shown in Figure 2c.

3. RECOVERY CODE GENERATION IN MULTILEVEL SPECULATION

3.1 Check Instructions and Recovery Code Representation for Multilevel Speculation

Multilevel speculation refers to the data speculation that occurs in a multilevel expression tree. One of a typical example of multilevel speculation is shown in the Figure 3a. It is also referred to as a *cascaded speculation* in Ju et al. [2000]. If $^{**}q$ is a potential weak update *only* to $^{*}p$, then the check instruction and recovery block will be as shown in Figure 3b. The expression $^{**}q$ could also be a potential weak update to both $^{*}p$ and $^{**}p$. We need to generate two check instructions for both $^{*}p$ and $^{**}p$, as shown in Figure 3c, each with a different recovery block. According to our previous study [Chen et al. 2002, 2004], there are frequent multilevel indirect references (e.g., multilevel field accesses) in the CPU2000 C programs. Those references present rich opportunities for multilevel speculative PRE.

The check and recovery code representation discussed in Section 2 can be applied directly to multilevel speculation without any change. In Figure 4b, only $^{*}p$ may be modified by the potential weak update. In Figure 4c, both the address expression $^{*}p$ and the value expression $^{**}p$ are candidates for speculative register promotion. Hence, two *if-block*s are generated. It is obvious that the

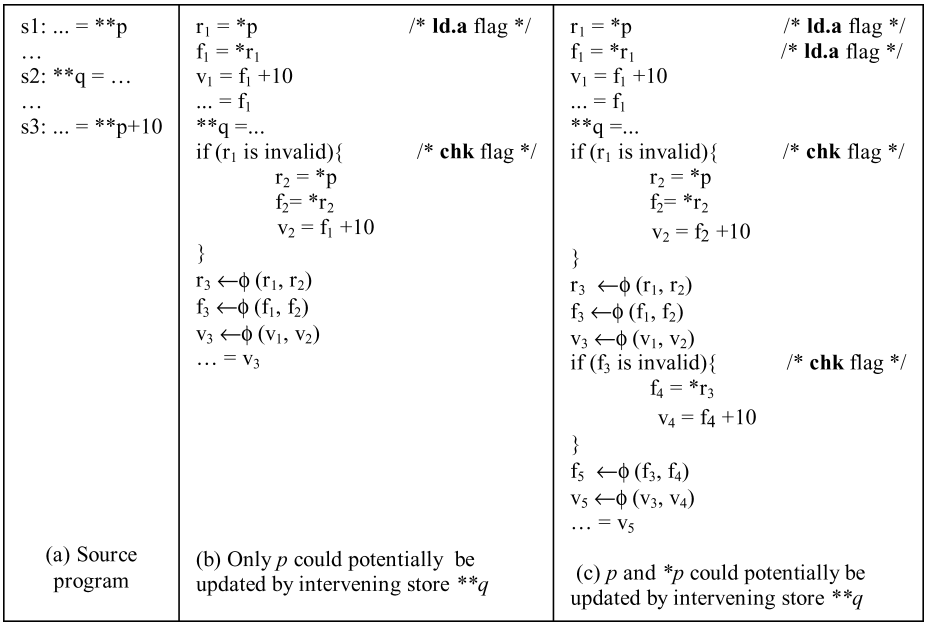

Fig. 4. Examples of check instructions and their recovery blocks in multilevel speculation.

representation in Figure 4 matches very well with final assembly code shown in Figure 3.

3.2 Recovery Code Generation for Multilevel Speculation

In this section, we will show how to integrate speculation support into partial redundancy elimination (PRE). There are two main partial redundancy elimination schemes: one is bit-vector based [Knoop et al. 1992] and the other is SSA form based [Kennedy et al. 1999]. In this paper, we focus on SSA form-based PRE (SSAPRE), because it is used in Intel's ORC compiler and our implementation is based on ORC.

There are six steps to identify redundant expression in SSAPRE: (1) Phi-insertion, (2) Rename, (3) DownSafety, (4) WillBeAvail, (5) Finalize, and (6) CodeMotion. The first two steps are aimed to identify the expressions that have the same value and are redundant. The following step 3 and step 4 are intended to handle the partial redundancy in the control flow graph. The Finalize step determines the placement of expressions. The last step, the CodeMotion step, transforms the code. More details can be found in Kennedy et al. [1999], and Lin et al. [2003].

In our speculative SSAPRE framework, an expression tree is processed in a bottom-up order. For example, given an expression $^{***}p$, we begin by processing the subexpression $^{*}p$ for the first-level speculation, then the subexpression $^{**}p$ for the second-level speculation, and last, $^{***}p$ for the third-level speculation. This processing order also guarantees that the placement of check instructions and the recovery code generation are optimized level-by-level. Therefore, when a subexpression is processed for the nth level speculation, every

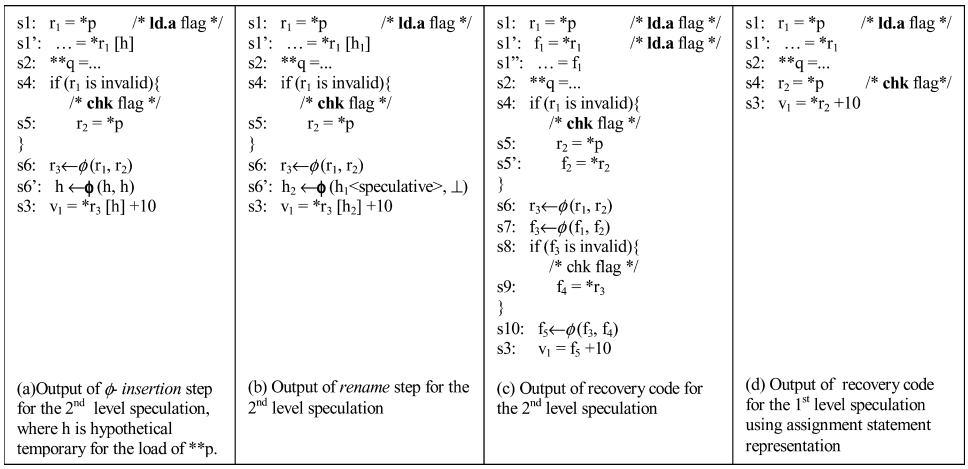

Fig. 5. Examples of recovery code generation in speculative PRE.

subexpression at the $(n-1)$th level speculation has been processed, and its corresponding recovery block (i.e., *if-block*) also generated. Here, we assume an *if-block* will be generated at the last weak update of the subexpression. A reload instruction for the value of the subexpression will be included initially in the *if-block*.

Figure 1c shows the result after the first-level speculation on $^*p$ for the program in Figure 1a. We assume $^{**}q$ is aliased with both $^*p$ and $^{**}p$ with a very *low* probability. Hence, $^{**}q$ in s2 is a speculative weak update for both $^*p$ and $^{**}p$. During the first-level speculation, the subexpression $^*p$ is speculatively promoted to the register r. The subscript of r represents the *version number* of r. As can be seen in Figure 1c, the value of $^*p$ in s1′ becomes unavailable to the *use* in s3 along the *true* path of the *if-block*, because of the redefinition from the reload instruction in s5. This is represented in the version number of r that changes from r_1 to r_3 because of the $\phi(r_1, r_2)$ operation in s6.

In the second-level speculation for $^{**}p$, we can also speculatively promote $^{**}p$ to the register f. A *hypothetical variable* h is created for $^*r$ (i.e. $^{**}p$) in s1′. Figure 5a shows the result after the ϕ-*insertion* phase in SSAPRE [Chow et al. 1997]. A ϕ-function $\phi(h, h)$ in s6′ is needed, because of the *if-block* in s4. The first operand of $\phi(h, h)$, which corresponds to the *false* path of the *if-block* in s4, will have the same version number as that in s1′, because of the speculative weak update in s2 after the *rename* step in SSAPRE [Chow et al. 1997]. However, the second operand of $\phi(h, h)$, which corresponds to the *true* path, is replaced by $\perp$, because the value of h (i.e. $^*r$) becomes *unavailable* due to the redefinition of r in s5. The result is shown in Figure 5b.

The *if-block* in s8 is inserted after the speculative weak update of s2 with a reload of $^*r$ in s9 (Figure 5c). Because of the second operand of $\phi(h_1, \perp)$ in s6′, the algorithm will insert a reload of $^*r$ (in s5′) along the *true* path of the *if-block* (in s4) to make the loading of $^*r_3$ in s3 fully redundant. That is, the existing SSAPRE algorithm will automatically update the recovery block from the first level (i.e., $^*p$) without additional effort.

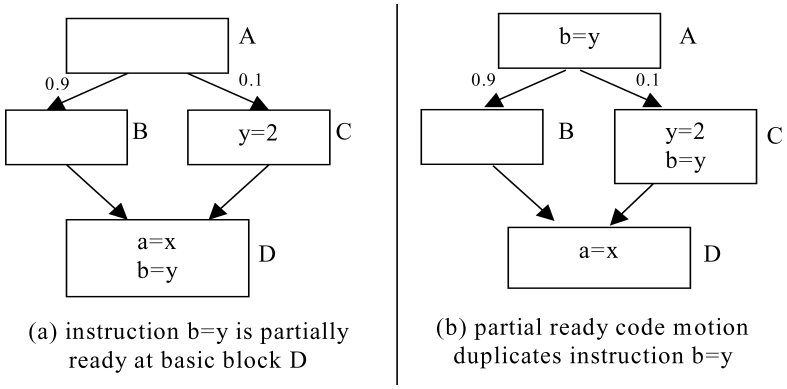

Fig. 6. Example of partial ready code motion for latency hiding.

This advantage of the *if-block* representation for the recovery block can be better appreciated if we look at the difficulties we would have encountered if the check instruction is represented as an assignment statement. The expressions $^{*}p$ and $^{**}p$ are speculative redundant candidates. Using an assignment statement to represent a check instruction, the recovery block generated for the expression $^{*}p$ is shown in Figure 5d. Since the value $^{*}r$ is redefined at s4, the compiler cannot detect the expression $^{*}r$ (i.e., $^{**}p$) in s3 as speculatively redundant (because r has been redefined). In order to support multilevel speculation, we would have to significantly modify the SSAPRE representation and thus increase the complexity of the algorithm substantially.

4. INTERACTION OF THE EARLY INTRODUCED RECOVERY CODE WITH LATER OPTIMIZATIONS

The recovery blocks, represented by *highly biased if-then* statements can be easily maintained in later optimizations, such as instruction scheduling. In this section, we illustrate this point using the *partially ready* code motion algorithm [Bharadwaj et al. 1999] as an example. Consider the example in Figure 6a, if we assume that the *right* branch is *very rarely* taken, the instruction $b = y$ in the basic block D can be identified as a P-ready (i.e., partial ready) candidate. It will then be scheduled along the *most likely* path $A \rightarrow B \rightarrow D$, instead of the unlikely path $A \rightarrow C \rightarrow D$. Figure 6b shows the result of such a code motion. The instruction $b = y$ is hoisted to the basic block A and a compensation copy is placed in the basic block C.

The recovery code represented by an *explicit* and *highly biased if-then* statement is a good candidate for P-ready-code motion. If there is a data-dependent instruction that could also be hoisted before the recovery block, this instruction will be duplicated as a compensation instruction in the recovery block. At the end of instruction scheduling, the recovery block would have been updated accordingly and is already well formed. There is no need to trace the flow dependence chain and generate the recovery block in a separate effort as in the JNMW scheme. After instruction scheduling, we could easily convert these *if-blocks* into check instructions and their respective recovery blocks.

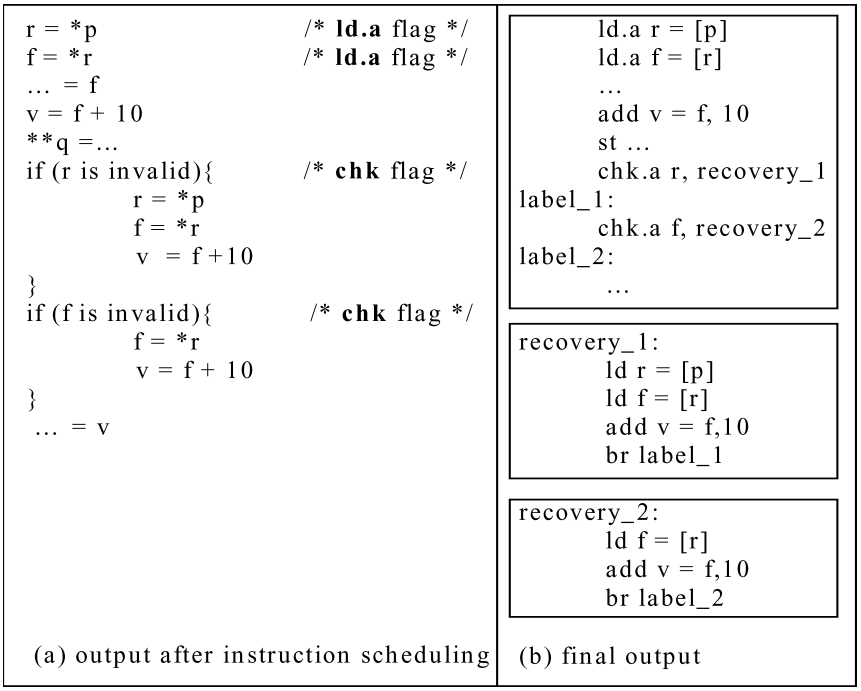

Fig. 7. Recovery code generation for a speculative PRE after instruction scheduling.

Using the example in Figure 1a, the output code of the speculative PRE (shown in Figure 5c) is further optimized with instruction scheduling. As can be seen in Figure 7a, if the compiler determines the instruction $v = f + 10$ can be speculatively hoisted across the store in the $^{**}q$ statement, it will duplicate the instruction in the recovery block. The generated check and recovery code are shown in Figure 7b.

Similar to that in speculative PRE, the proposed scheme can also be used in speculative instruction scheduling for both control and data speculation. We use the example in Figure 2a to show how to use partial ready-code motion to achieve the effect of multilevel control speculation. Figure 8a shows the intermediate representation after the compiler selects the load instruction $^{*}r$ as the candidate of partial ready-code motion and move $^{*}r$ out of the branch. The final code generation phase converts these two special *if* statements into *chk.s* instructions and generates the corresponding recovery block as shown in Figure 8b.

5. OPTIMIZING THE PLACEMENT OF CHECK INSTRUCTIONS AND RECOVERY BLOCKS IN SPECULATIVE PRE

The newly generated check instructions and recovery blocks after the speculative PRE may introduce unnecessary checks. For example, the algorithm described in Section 3.2 that inserts a *check instruction* after the *last weak update* may be unnecessary on some execution paths if the *uses* are in the branch basic block while check instructions are generated at the control-flow merge points.

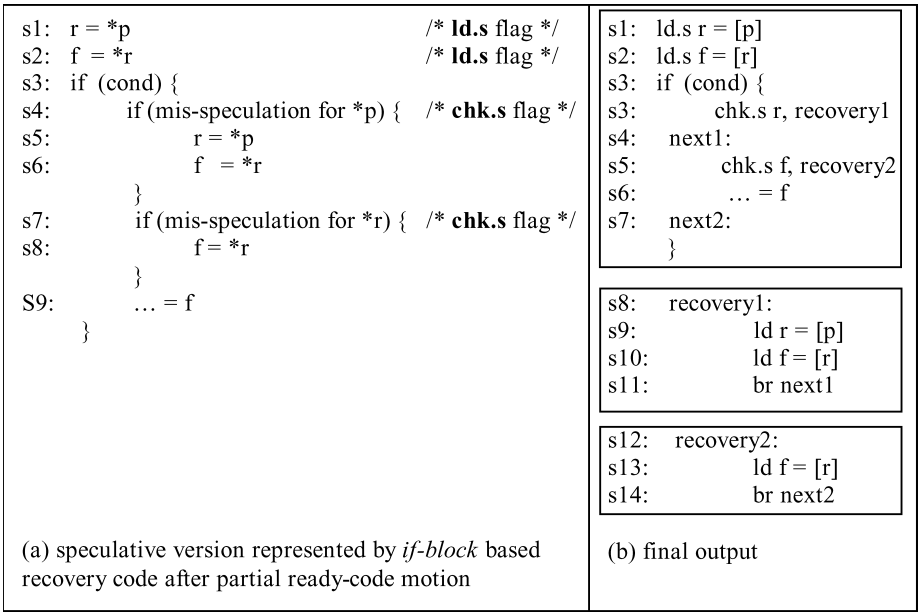

Fig. 8. Recovery code generation for multilevel control speculation using partial ready-code motion.

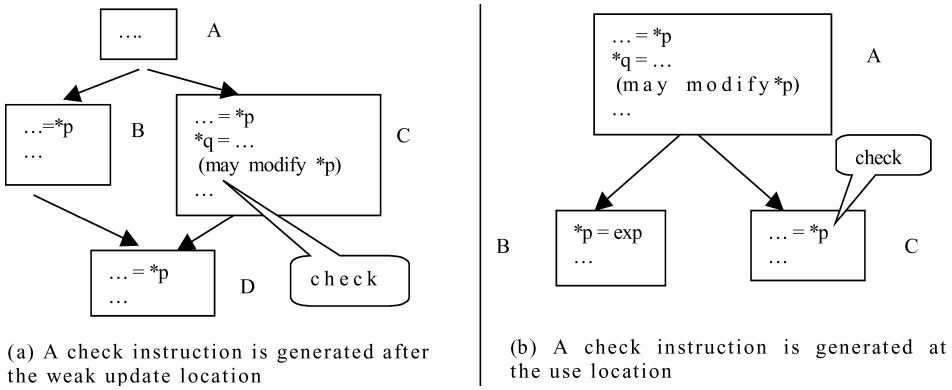

Fig. 9. Examples of effective placement of check instructions in speculative PRE.

Those extraneous check instructions may degrade performance. In this section, we examine the issues regarding how check instructions should be placed in order to minimize the execution of unnecessary check instructions.

There are two obvious locations to place a check instruction during the speculative PRE optimization: one is at the *use* location of the speculative redundant expression. The other is *after* the *last weak update* but *before* the use of the speculative load as we already mentioned. Both schemes can introduce unnecessary check instructions.

Consider the example in Figure 9a. The occurrence of expression $^*p$ in basic block D is assumed to be *speculatively redundant* to $^*p$ in basic block C, and *nonspeculatively redundant* to $^*p$ in basic block B. In this case, it is better to

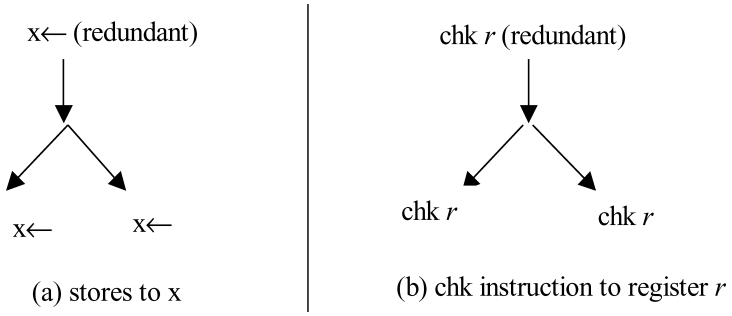

Fig. 10. Store redundancy versus check instruction redundancy.

generate a check instruction after the *weak update* *q in basic block C than a check instruction at the *use* location in basic block D. This is because the check instruction is unnecessary if the program executes along the path $A \rightarrow B \rightarrow D$.

The example in Figure 9b shows a different situation. It is better to generate a check instruction at the *use* location because a check instruction generated *after* the *weak update* *q will be unnecessary if the path $A \rightarrow B$ is executed. It is difficult to determine which check instruction placement scheme is superior, because it depends on the structure of the application program.

5.1 Check Placement Optimization

Let us first consider the scheme that places check instructions *after* the *last weak update* point. We can eliminate unnecessary check instructions under the framework of *partial store redundancy elimination* (PSRE). A similar algorithm based on *partial load redundancy elimination* (PLRE) can be developed if check instructions are inserted at the use location. It is recognized that PLRE is a *dua*l framework to PSRE and, hence, we will describe only the PSRE approach here. In order to simplify our discussion, we model *if-block*-based recovery code in the form "*if (r is invalid)* {. . . }" as *chk r*.

We perform PSRE using the static single use (SSU) form [Lo et al. 1998].[1] A *store* of the form x←r is fully (partially) redundant if the *store* is *fully (partially) anticipated*. A *store* is *anticipated* if the *store* is never used before it is redefined, or reaches an exit of the block. As shown in Figure 10a, the first occurrence of the *store* is *anticipated* and thus it is a redundant *store*. Similarly, a check instruction in the form of *chk* r is fully (partially) redundant if the check instruction is *fully (partially) anticipated*. Thus, given two occurrences of a check instruction *chk r*, if there is no intervening *use* of the register r, the earlier check instruction is redundant as shown in Figure 10b. The anticipation of a check instruction is killed when the register r is used. The movement of a check instruction during the code motion phase is blocked by a *use* or a *definition* of the register r.

In our redundant check instructions elimination, a check instruction corresponds to a *store*, and a *use* of register r corresponds to a *load*. The reader is referred to Lo et al. [1998] for a full discussion of SSUPRE.

[1]The SSU form is adopted in Intel's ORC compiler [Ju et al 2001], which is the basis of our approach.

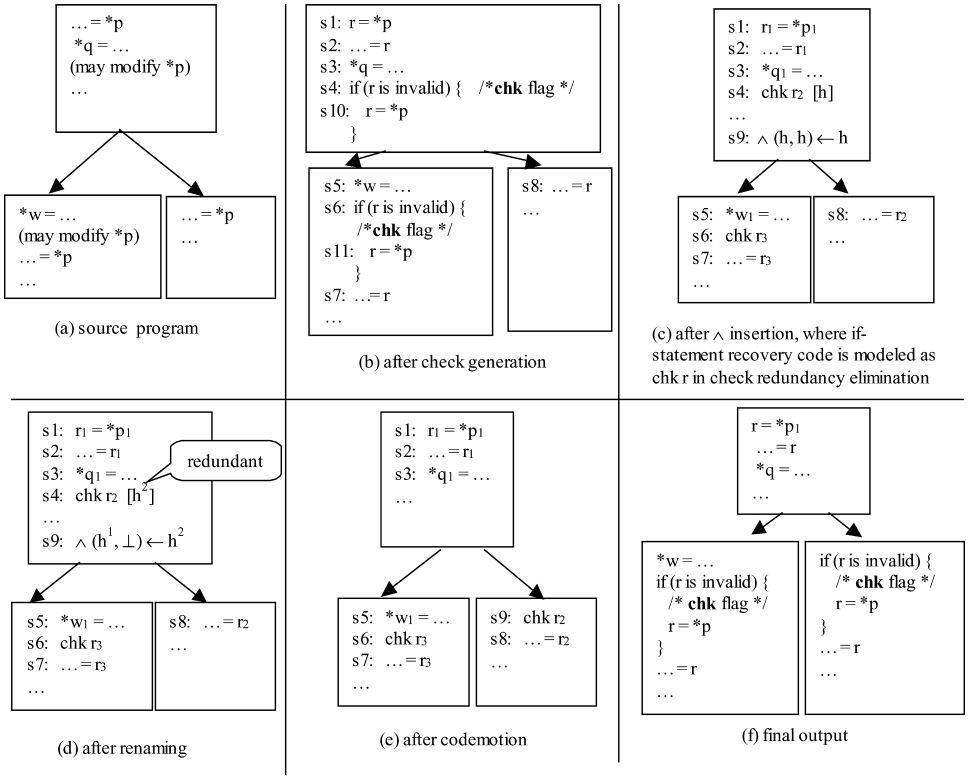

Fig. 11. A running example to show the effect of check placement optimization (h is the hypothetical temporary for *chk r)*.

5.2 The ∧-Insertion Step

The SSU form we use here is a dual of SSA form. Just like the ϕ operator is regarded as a *definition* of the corresponding variable and always defines a new version, the $\wedge$ operator is regarded as a *use* of variable and always establishes (uses) a new version. The ϕ insertion occurs at a merge point while the $\wedge$ insertion occurs at a split point. The purpose of $\wedge$ insertion is to expose the potential insertion points for the check instruction being optimized. There are two situations that cause $\wedge$'s to be placed. The first situation is when we encounter a check instruction, we insert a $\wedge$-*function* at its *iterated postdominance frontiers* [Lo et al. 1998]. The second situation is when a *use* or a *definition* of register r reaches a split point. Since a check instruction can be *killed* by a *use* of register r, this means $\wedge$'s have to be placed at the *iterated postdominance frontier* of each *use* or *definition* of register r.

Consider the example code in Figure 11a. Figure 11b is the result of the program after one *if-block*-based check is placed at the point of the *weak update* at s6 for the *use* of the expression $^*p$ in s7, and another check is generated at s3 for the *use* of $^*p$ in s8 In order to eliminate the check effectively, the *if-block*-based recovery code is modeled as a check instruction in the form of *chk* r as shown in Figure 11c at the $\wedge$ insertion phase in7 the check placement

optimization. In Figure 11c, a $\wedge$-*function* at the control flow split is inserted at s9 because the *use* of register r in s7 and s8 reaches the split.

5.3 The Rename Step

The purpose of the *Rename* step is to assign SSU versions to all of the check instructions. Each *use* of register r is assigned a new SSU version. The check instructions that reach the *use* will have the same version numbers. Each $\wedge$-*function* is also assigned a new SSU version because we regard each $\wedge$-*function* as a *use*.

Renaming is performed by conducting a preorder traversal of the postdominator tree, beginning at the exit points of the program. We maintain a renaming stack for each check instruction in the program. When we came across a *use* or a *definition* of register r or $\wedge$, we generate a SSU version and push it onto the stack. When we come to a check instruction, we assign it the SSU version currently at the top of its stack, and also push it onto the stack. The operands of $\wedge$ are assigned the SSU version at the top of its stack if the top is a check or a $\wedge$; otherwise, it is assigned $\perp$. The renaming effect is illustrated in Figure 11d. The occurrence of *chk r* at s4 is assigned with the same version number as the one at s9. Thus, it is marked with a redundant flag.

The last step (i.e., CodeMotion step) performs the insertion and the deletion of check instructions. Any check instructions along the postdominator tree that are assigned the same SSU versions are redundant, and could be deleted. In Figure 11d, the first *chk r* can now be eliminated as shown in Figure 11e. It also inserts a *chk r* at statement s9 for the use at s8. The final output is shown in Figure 11f after the compiler eliminates the corresponding *if-block*-based recovery block for the first *chk r*, and insert one *if* statement for the newly generated *chk r*.

Note that if we use another scheme that places a check instruction at the location of a use, we could have a similar algorithm based on partial load redundancy elimination (PLRE). It will use a framework based on SSAPRE, since it is the dual of the SSUPRE algorithm. Because of the lack of space, we will not describe the scheme here.

6. PERFORMANCE MEASUREMENTS

We implemented our new check instruction and recovery code generation algorithm for general speculative PRE optimizations in Intel's ORC compiler version 2.0, and tested on the HP zx2000 workstation equipped with an Itanium-2 900MHz processor and 2 GB of memory. Ten CPU2000 programs are used in this measurement. The base versions used for comparison are compiled at optimization level O3. The control speculation is enabled by default in the instruction scheduling.

Our experiments contain three parts. In the first part, we study the impact of representing the check instruction and the recovery code as an *if-block* on the code generated by the compiler and its run-time performance. In the second part, we evaluate the performance impact of the recovery code generation with the multilevel speculation support. In the last part, we study the performance

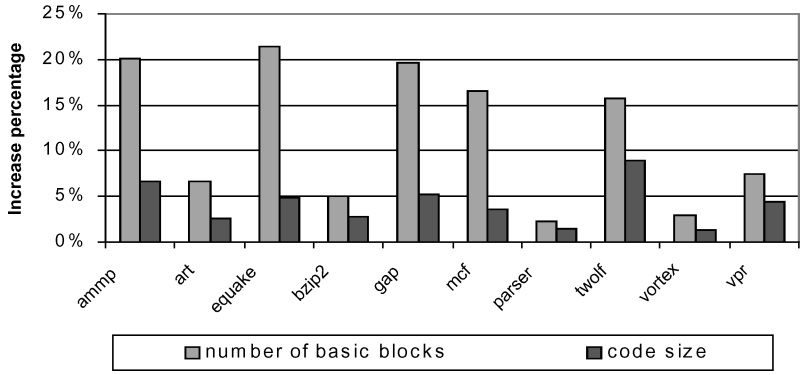

Fig. 12. Number of basic blocks and code size increase with the *if-block*-based recovery code generation scheme.

Table I. Percentage of Instruction Cache Miss Stalls With the *if-block*-Based Recovery Code Scheme

	ammp	art	equake	bzip2	gap	mcf	parser	twolf	vortex	vpr
1[a]	0.3%	0.01%	0.05%	0.04%	8.6%	0.02%	0.4%	9.5%	0.03%	3.1%
2[b]	0.3%	0.02%	0.06%	0.05%	8.9%	0.03%	0.5%	10.6%	0.04%	3.3%

[a]The percentage of instruction cache miss stalls in the total CPU cycles in base version.
[b]The percentage of instruction cache miss stalls in the total CPU cycles after recovery code generation.

impact of redundant check elimination. The performance data is collected using the Pfmon tool [Pfmon 2003].

6.1 Impact of Recovery Code Generation

Our recovery code generation algorithm is quite effective in practice. We show the increase in the number of basic blocks and the code size of the ten benchmarks from the speculative PRE optimization in Figure 12. We collected the number of basic blocks at the compilation time and the code size is measured using the size of text segments. The inserted *ld.c* and/or *chk.a* instructions and respective recovery blocks contribute only slightly to the static code size. Note that since our general speculative optimizations can eliminate speculative redundant expressions, it would offset the code increase from the check instructions and their recovery blocks. As can be seen in Figure 12, the impact on code size is marginal except for *twolf*. We also collected the execution stall cycles because of instruction cache misses at runtime as shown in Table I. We observe that the impact from the increase of instruction cache misses is insignificant for most programs, except for *twolf*. *Twolf* has a relatively high instruction cache miss penalty to start with. Our recovery code generation makes it worse. However, the benefit from data speculation optimization in *twolf* still outweighs the performance loss from increased instruction cache misses.

Figure 13 shows performance comparison of recovery code generation in speculative PRE using the assignment representation, the *improved* assignment representation and the *if-block* representation. The main difference between the assignment-based scheme as mentioned in Section 3.2 and the *improved*

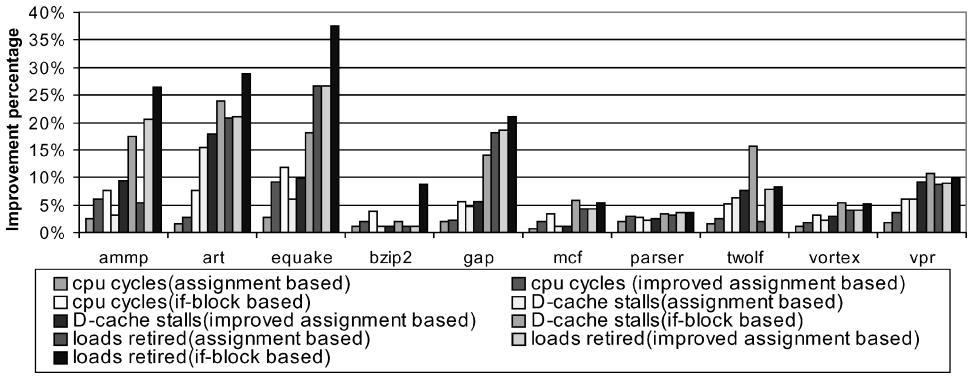

Fig. 13. Overall performance comparison in speculative PRE using assignment based scheme and the *if-block* based scheme, where D-cache stalls are stalls caused by data-load latencies and other memory system overhead.

assignment-based scheme is that we spent much effort to modify later analysis and optimization phases to minimize the negative impact from the assignment statements generated from the check instructions. For example, we used some extra flags on these special assignment statements so that the later data flow analysis can ignore this kind of special definitions. Otherwise, it may lead to some unexpected renaming. We will illustrate the details in Section 7. Overall, Figure 13 shows that the *if-block*-based outperforms assignment-based schemes. The improvement comes from better optimizations in later phases. For the assignment-based approach, we have to help the compiler to recognize the speculative code. For some benchmarks, such as *parser* and *bzip2*, the performance difference is rather small. This is because the opportunity for data speculation in these two benchmarks is low. Also, despite our numerous efforts in minimizing the negative impact of assignment statements introduced by check instructions, the improvement in performance with the *improved* assignment-based is still lower than the *if-block*-based scheme. However, we should not overlook the key contribution of the *if-block* representation that avoids the tedious performance tuning required in later analysis and optimization phases to overcome the inadequacy of the assignment based representation of check instructions.

6.2 Performance Comparison with Single- and Multilevel Data Speculation

Figure 14 shows the performance difference in speculative PRE with single- and multilevel speculation support. We can observe some performance difference for benchmarks *ammp*, *equake*, and *gap* when multilevel data speculation is enabled. Overall, the impact on performance is not substantial. Consider *Equake*, for example. Many intervening *stores* exist between multiple redundant multilevel memory *loads*. Since multilevel memory references here are of the pointer type and the *type-based alias analysis* assumes that aliases could only occur among memory references of the *same* type, hence, the possible aliasing caused by intervening *stores* can be ignored. However, multilevel speculation opportunities can still be applied even when the type-based alias analysis is enabled.

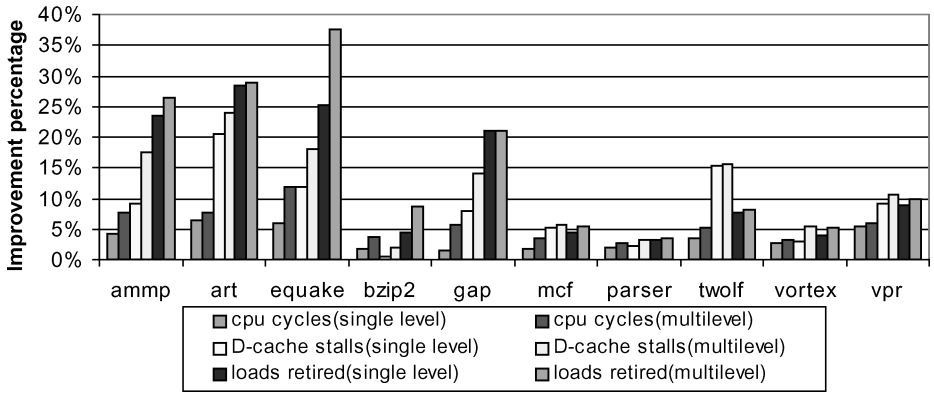

Fig. 14. Performance comparison using *if-block*-based recovery code generation between single- and multilevel data speculation support, where the base version used for comparison is compiled with –O3.

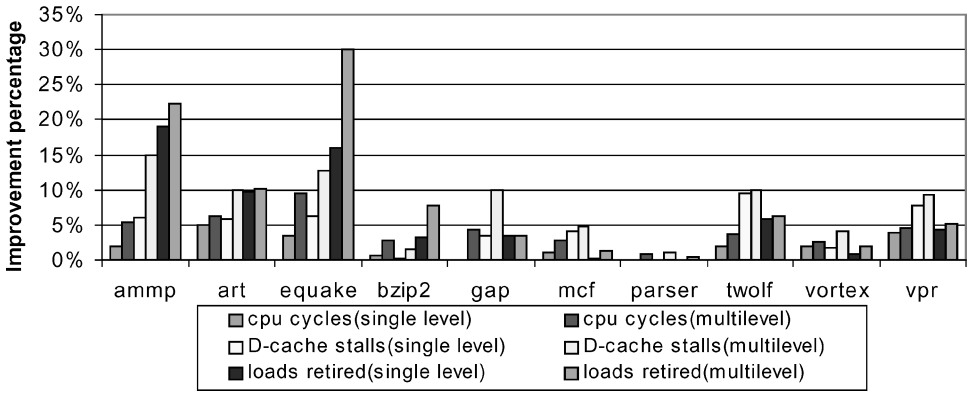

Fig. 15. Performance comparison using *if-block*-based recovery code generation between single- and multilevel data speculation support, where the base version used for comparison is compiled with –O3 and type-based alias analysis.

In the code segment shown in Figure 16a, the types of expression *w[col][1], A[Anext]*, and *A[Anext][0]* are different, so the compiler could assume that there is no alias existed among expressions *w[col][0]*, *A[Anext]*, and *A[Anext][1]*. We can see that expressions *A[Anext]* and *A[Anext][1]* can be promoted to registers simply using type-based alias analysis. Nevertheless, the opportunity of data speculation can still be applied to those pointer references of the same type. For example, the expression *A[Anext][1][1]* cannot be promoted to a register, because it is of the same type as the *store* expression *w[col][0]*. In Figure 15, we can see that both single- and multilevel speculation increase performance for some benchmarks, even when type-based alias analysis is enabled.

Figure 16b shows another example that the type-based alias analysis would fail when the intervening *stores* have the same type as the multilevel memory *loads* or *computation expressions*. In Figure 16b, the expression *Src.rake*, *Src.strike* and *Src.dip* are of the same type as the expressions $^*u$ and $^*v$. Therefore, with single-level data speculation, our compiler can speculatively promote

```
for (i = 0; i < nodes; i++) {
  . . .
  while (Anext < Alast) {
    col = Acol[Anext];
    sum0 += A[Anext][0][0] *...
    sum1+= A[Anext][1][1] *...
    sum2+= A[Anext][2][2] *...
    w[col][0] += A[Anext][0][0]*v[i][0] + ...
    w[col][1] += A[Anext][1][1]*v[i][1] + ...
    w[col][2] += A[Anext][2][2]*v[i][2] + ...
    Anext++;
  }
 }
}
```

(a) Multilevel speculation for multiple level memory loads

```
*u = (cos(Src.rake) * sin(Src.strike) -
      sin(Src.rake) * cos(Src.strike) * cos(Src.dip));
*v = (cos(Src.rake) * cos(Src.strike) +
      sin(Src.rake) * sin(Src.strike) * cos(Src.dip));
*w = sin(Src.rake) * sin(Src.dip);
```

(b) Multilevel speculation for computation expressions including intrinsic procedure call such as *sin* and *cos*

Fig. 16. Two examples of multilevel speculation opportunities in benchmark *Equake*.

expressions *Src.rake*, *Src.strike*, and *Src.dip* into registers. With multilevel speculative PRE, our compiler can further speculatively promote the values of expressions *sin(Src.rake), sin(Src.strike), cos(Src.rake), cos(Src.strike)* and *cos(Src.dip)* to registers. This would effectively eliminate six expensive function calls. We think that such performance opportunity may exist in some other important applications. Finally, it should be noted that the type-based alias analysis is not always safe to be applied to applications. In the future, we will further investigate the impact of multilevel speculation on the recovery code generation using more application programs.

6.3 Performance Impact of Recovery Code Redundancy Elimination

Redundant recovery code elimination could improve the performance of speculative optimizations. The decrease of check instructions can reduce the execution time of those unnecessary check instructions. More importantly, since the Advance Load Address Table (ALAT) on Itanium uses partial address for address matching, those unnecessary check instructions may lead to mis-speculations. In our framework, we apply the redundant recovery code elimination for *if-block*-based scheme with single- and multilevel speculation support.

Figure 17 shows the decrease in the number of check instructions at compilation time after redundant check elimination is applied to the recovery code generation algorithm with single- speculation and multilevel speculation support. We can see that the redundant check elimination can reduce some check instructions for the benchmarks. For benchmarks *equake*, *bzip2*, and *mcf*, the opportunities for redundant check elimination under multilevel speculation are not as good as other programs.

In Figure 18, we measure the actual performance impact by redundant check elimination. For benchmarks *ammp*, *art*, and *gap*, multilevel speculation can benefit more from redundant check elimination. The reason is that redundant check elimination can reduce more *chk.a* instructions (instead of *ld.c* instructions) with multilevel speculation support. The overhead of *chk.a* instruction is much higher than that of *ld.c* instruction. We also show the decrease in the mis-speculations. Overall, the mis-speculation ratio for these ten benchmarks is

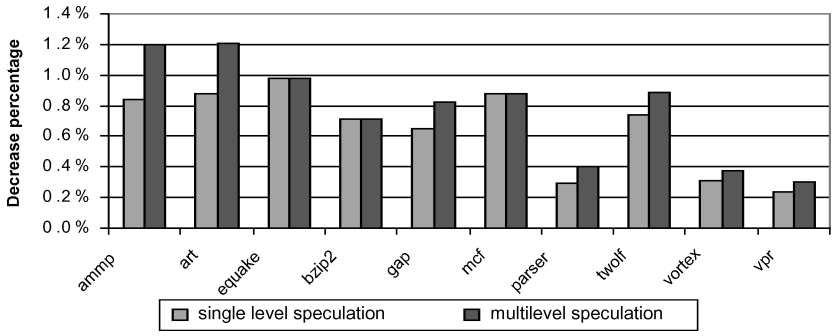

Fig. 17. Decrease in the number of check instructions at compilation time under *if-block*-based recovery code generation framework with single- and multilevel speculation support.

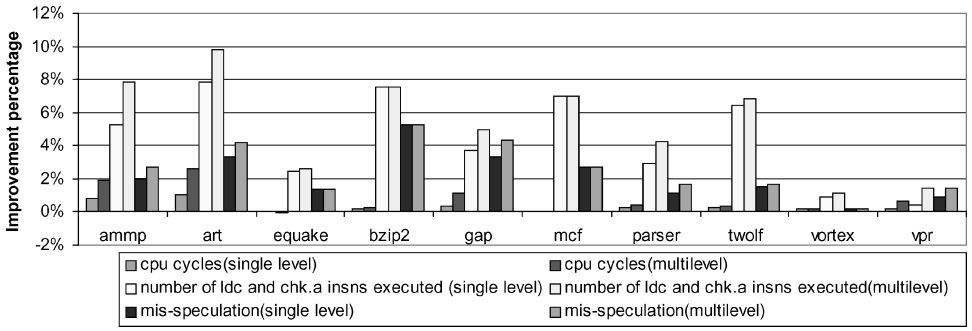

Fig. 18. Performance impact of redundant check instructions elimination under *if-block*-based recovery code generation with single- and multilevel data speculation support.

very low (below 1%). Therefore, the decrease in mis-speculation is insignificant for the overall performance.

7. RELATED WORK

Most of the recovery code generation schemes proposed so far only deal with specific optimizations. If the expression $^*p$ and $^*q$ are potential aliases with a very*low* probability, the *load* instruction of one expression can be speculatively moved across the *store* instruction of another expression. A *chk.a* instruction is inserted at its original load location to check for possible mis-speculation at runtime. The check instruction will jump to a recovery block if a mis-speculation does occur.

To facilitate correct code scheduling, a dependence edge is added from the *ld.a* instruction to the added *chk.a* instruction. The dependence edge is shown as a dotted edge in Figure 19c. This dependence ensures that the *chk.a* instruction is always scheduled after the *ld.a* instruction. A dependence edge from *chk.a* to *st [r]* and another from *st [q]* to *chk.a* are also added to enforce correct code scheduling. Since the instruction $add\ r3 = r1, r2$ only depends on $ld.ar1 = [p]$, it can be speculatively moved across *st [q]*, as shown in Figure 19d. During the recovery code generation phase, the compiler will place the original *load* instruction, i.e., $ldr1 = [p]$ and *all* of the instructions between *ld.a* and

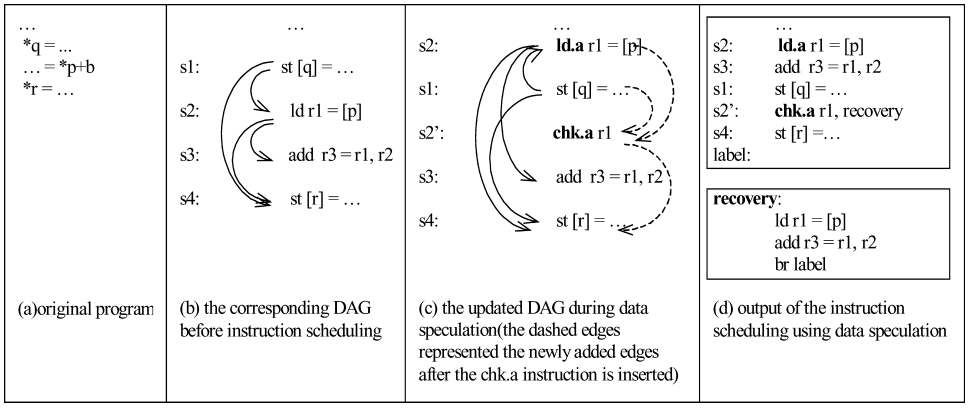

Fig. 19. Example of the JNMW recovery code generation algorithm used for instruction scheduling.

chk.a that are *flow dependent* on *ld.a* instruction, i.e., $add\ r3 = r1, r2$, into the recovery block (see Figure 19d).

This algorithm works well for code scheduling, but not suitable for more general speculative optimizations, such as the speculative partial redundancy elimination optimization. Consider the example in Figure 20a, and assuming $^*q$ and $^*p$ are unlikely to be aliases, the compiler can speculatively replace the expression $^*p+b$ (s4) by $a+b$, (s1) as shown in Figure 20c. Using the JNMW algorithm, the later recovery code generation phase may not locate the instruction on the dependence chain between the *ld.a* and the *chk.a*, because it has been speculatively eliminated. This can be shown with the output in Figure 20(b), where the instruction $^*p + b$ (i.e., $add\ r3 = r4, r2$ in Figure 20c) is eliminated. Keeping such *eliminated* instructions on the dependence chains in the IR can significantly complicate later optimizations because they must be handled differently from regular instructions.

Another difficulty with the JNMW algorithm is that it implicitly assumes there is *only one* speculative load (*ld.a*) instruction for each *chk.a*. This assumption is no longer true for speculative PRE optimizations. For speculative PRE, one *chk.a* instruction may correspond to multiple *ld.a* instructions, so multiple-flow dependence chains should exist for a single *chk.a* instruction.

This can be shown with the example in Figure 20d. We can speculatively eliminate $^*p$ (in s6) and use the value loaded in s1 or s3. A *ld.a* instruction is generated for s1 and s3, respectively (as shown in Figure 20e). The *chk.a* instruction in s6 corresponds to multiple *ld.a* instructions (at s1 and s3). If the compiler selects the *flow-dependence chain*, based on the *ld.a* instruction in s3, according to the JNMW algorithm, the statement s4, i.e., *add r2 = r1, 1*, would be included in the recovery block. However, this instruction is *nonspeculative*, and should not be included in the recovery code. The correct recovery block should be as shown in Figure 20g. This example shows the need to distinguish nonspeculative instructions from speculative instructions in IR for correct recovery code generation. All later optimization phases must be aware of the existence of possible recovery blocks and check instructions.

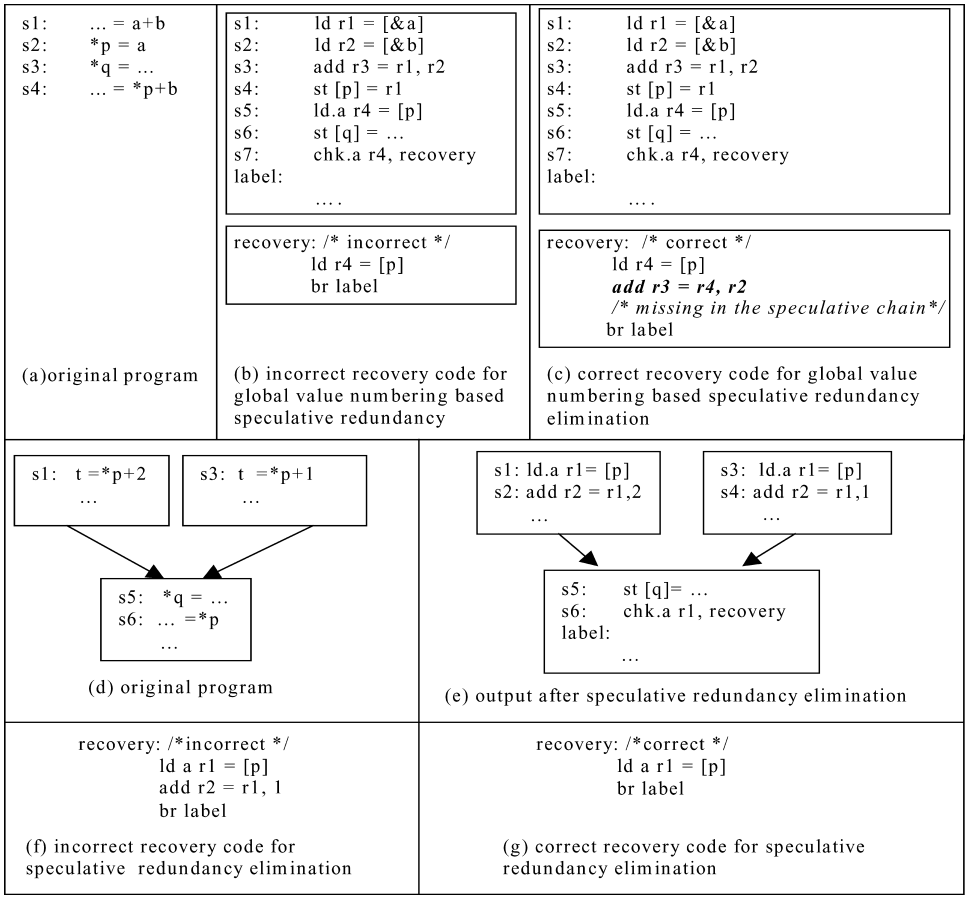

Fig. 20. Examples to show that the scheme proposed in Ju et al. [2000] is insufficient to handle general speculative optimizations. (a), (b), and (c) show one situation where some instructions could be missing in the recovery block; (d), (e), and (f) show another situation where some instructions could be mistakenly added into the recovery block.

In Lin et al. [2003, 2004], a check instruction is represented *implicitly* as an*assignment statement* and its recovery block as its associated flow-dependence chain. Consider a simple example in Figure 1a. The second occurrence of expression $^*p$ in s3 can be regarded as *speculatively redundant* to the first $^*p$ in s1. Therefore, we can use *ld.a* for the first load of $^*p$, and use the check instruction for the second $^*p$. In Figure 5d, an *assignment statement* shown as $r_2 = *p$ with a *check* flag is used to represent *chk.a* or *ld.c*, respectively. The recovery block is *implicitly* assumed to have the original *load* instruction for $^*p$ in s3. In later code scheduling, if some instructions can be speculatively moved up, as the example in Figure 19, the flow dependence chain between *ld.a* and the check instruction *implicitly* represents the remaining code inside the recovery block. However, as shown in Figure 5d, the *live range* of the register r has been changed and the first definition of r in s1 can no longer reach the last use of $r$in s3. The version number of r is changed in s4 from $r1$ to $r2$ because

the *check* instruction is represented as an *assignment* statement. This, in some sense, violates the *semantics* of the *check* instruction that requires both the *speculative load* and the corresponding *check instruction* to use the *same* register. In addition, this *implicit* representation could significantly inhibit later optimizations, such as code scheduling, for instructions related to the speculative load. For example, it no longer allows the instruction $r + 10$ in s3 to be speculatively hoisted across the *store* instruction $^*q$ in s2. Existing check and recovery code generation approaches are thus inadequate for general speculative optimizations.

8. CONCLUSIONS

In this paper, we propose a compiler framework to generate check instructions and recovery code for *general* speculative optimizations. The contributions of this paper are as follows: First, we propose a simple *if-block* representation for the check instructions and their corresponding recovery blocks to support both control and data speculation. It allows speculative recovery code introduced early on during program optimizations to be seamlessly integrated with subsequent optimizations. We use this framework to generate the recovery code for speculative PRE-based optimizations that include partial redundancy elimination, register promotion and strength reduction. We then show that the recovery code generated for speculative PRE can be integrated seamlessly with later optimizations such as instruction scheduling. This recovery code representation can also be applied to speculative instruction scheduling. Furthermore, we study the optimization that eliminates unnecessary check instructions. We model the unnecessary check elimination as a partial store redundancy elimination (PSRE) problem and use the static single-use (SSU) form to effectively solve it. Finally, we show that our proposed framework can support the compiler to generate recovery code for both single and multilevel speculations. We have implemented the proposed recovery code generation framework in the Intel's ORC compiler.

As for future work, we would like to support more speculative optimizations, such as value speculation and thread-level speculation, under the same framework to ensure its generality and to exploit additional performance opportunities.

ACKNOWLEDGMENTS

The authors wish to thank Sun Chan (Intel), Peng Tu (Intel), Raymond Lo for their valuable suggestions and comments.

This work was supported in part by the U.S. National Science Foundation under grants EIA-9971666, CCR-0105571, CCR-0105574, and EIA-0220021, and grants from Intel.

REFERENCES

BHARADWAJ, J., MENEZES, K., AND MCKINSEY, C. 1999. Wavefront scheduling: Path based data representation and scheduling of subgraphs, *MICRO'32*, (Dec.).

BRINGMANN, R. A., MAHLKE, S. A., HANK, R. E., GYLLENHAAL, J. C., AND HWU, W. W. 1993. Speculative execution exception recovery using write-back Suppression. *MICRO'26*.

CHEN, T., LIN, J., HSU, W., AND YEW, P. C. 2002. An empirical study on the granularity of pointer analysis in C Programs. In *15th Workshop on Languages and Compilers for Parallel Computing*. 151–160.

CHEN , T., LIN, J., HSU, W., AND YEW, P. C. 2004. Data dependence profiling for speculative optimization. In *Proceedings of the 14th International Conference on Compiler Construction*. 57–62.

CHOW, F., CHAN, S., LIU, S., LO, R., AND STREICH, M. 1996. Effective representation of aliases and indirect memory operations in SSA form. In *Proceedings of the Sixth International Conference on Compiler Construction*. 253–267.

CHOW, F., CHAN, S., KENNEDY, R., LIU, S., LO, R., AND TU, P. 1997. A new algorithm for partial redundancy elimination based on SSA form. In *Proceedings of the ACM SICPLAN Conference on Programming Language Design and Implementation*. 273–286.

CYTRON, R., FERRANTE, J., ROSEN, B., WEGMAN, M., AND ZADECK, K. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, *13*, 4, 451–490.

ISHIZAKI, K., INAGAKI, T., KOMATSU, H., AND NAKATANI, T. 2002. Eliminating exception constraints of Java programs for IA-64. In *Proceedings of International Conference on Parallel Architecture and Compiler Technology*. 259–268.

JU, R. D.-C., NOMURA, K., MAHADEVAN, U., AND WU, L.-C. 2000. A unified compiler framework for control and data speculation. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. 157–168.

JU, R. D.-C., CHAN, S., AND WU, C. 2001. Open Research Compiler (ORC) for the Itanium Processor Family. Tutorial. *MICRO 34*.

HEGGY, B. AND SOFFA, M. L. 1990. Architectural support for register allocation in the presence of aliasing. In *Proceeding of Supercomputing 1990*.

KAWAHITO, M., KOMATSU, H., AND NAKATANI, T. 2000. Effective null pointer check elimination utilizing hardware trap. In *Proceeding of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems.*

KENNEDY, R., CHOW, F., DAHL, P., LIU, S.-M., LO, R., AND STREICH, M. 1998. Strength reduction via SSAPRE. In *Proceedings of the Seventh International Conference on Compiler Construction*. 144–158.

KENNEDY, R., CHAN, S., LIU, S., LO, R., TU, P., AND CHOW, F. 1999. Partial redundancy elimination in SSA Form. *ACM Trans. on Programming Languages and Systems, 21*, 3, 627–676.

KNOOP, J., RUTHING, O., AND STEFFEN, B. 1992. Lazy code motion. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 224–234 .

LIN, J., CHEN, T., HSU, W. C., YEW, P. C., JU R. D.-C., NGAI, T. F., AND CHAN S. 2003. A compiler framework for speculative analysis and optimizations. In *Proceedings of ACM SIGPLAN on Programming Language Design and Implementation*. 289–299.

LIN, J., HSU, W. C., YEW, P. C., JU, R. D.-C., NGAI, T. F., AND CHAN, S. 2004. A compiler framework for recovery code generation in general speculative optimizations. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. 17–28.

LO, R., CHOW, F., KENNEDY, R., LIU, S., AND TU, P. 1998. Register promotion by sparse partial redundancy elimination of loads and stores. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 26–37.

MAHADEVAN, U., NOMURA, K., JU, R. D.-C., AND HANK, R. 2000. Applying data speculation in modulo scheduled loops. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. 169–176.

MAHLKE, S. A.,CHEN, W. Y., BRINGMANN R. A, HANK, R. E., HWU, W.-M., RAU, B. R., AND SCHLANSKER, M. S. 1993. Sentinel scheduling: a model for compiler-controlled speculative executions. *ACM Trans. Comput. Syst.*, *11*, 4, 276–408.

PFMON 2003. ftp://ftp.hpl.hp.com/pub/linux-ia64/pfmon-1.1-0.ia64.rpm.

POSTIFF, M., GREENE, D., AND MUDGE, T. 2000. The store-load address table and speculative register promotion. In *Proceedings of International Symposium on Micro Architecture*. 235–244.

WU, Y. AND LARUS, J. R. 1994. Static branch frequency and program profile analysis. In *Proceedings of the 27th Annual IEEE/ACM International Symposium on Micro Architecture*. 1–11.

Received February 2005; revised July 2005 and November 2005; accepted December 2005