A Compiler Framework for Speculative Optimizations

JIN LIN, TONG CHEN, WEI-CHUNG HSU, PEN-CHUNG YEW
and
ROY DZ-CHING JU, TIN-FOOK NGAI, SUN CHAN

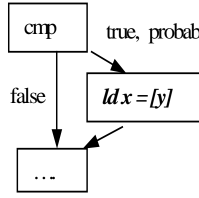
Speculative execution, such as control speculation or data speculation, is an effective way to improve program performance. Using edge/path profile information or simple heuristic rules, existing compiler frameworks can adequately incorporate and exploit control speculation. However, very little has been done so far to allow existing compiler frameworks to incorporate and exploit data speculation effectively in various program transformations beyond instruction scheduling. This paper proposes a speculative SSA form to incorporate information from alias profiling and/or heuristic rules for data speculation, thus allowing existing frameworks to be extended to support both control and data speculation. Such a general framework is very useful for EPIC architectures that provide runtime checking (such as *advanced load address table* (ALAT)) on data speculation to guarantee the correctness of program execution. We use SSAPRE as one example to illustrate how to incorporate data speculation in partial redundancy elimination (PRE), register promotion, and strength reduction. Our extended framework allows both control and data speculation to be performed on top of SSAPRE and, thus, enables more aggressive speculative optimizations. The proposed framework has been implemented on Intel's Open Research Compiler (ORC). We present experimental data on some SPEC2000 benchmark programs to demonstrate the usefulness of this framework.

Categories and Subject Descriptors: D3.4 [**Programming Languages**]: Processors – *Compiler; Optimization*;
General Terms: Algorithms, Performance, Design, Experimentation
Additional Key Words and Phrases: Speculative SSA form, speculative weak update, data speculation, partial redundancy elimination, register promotion

1. INTRODUCTION

Data speculation refers to the execution of instructions on *most likely correct* (but *potentially incorrect*) operand values. *Control speculation* refers to the execution of instructions *before* it has been determined that they would be executed in the normal flow of execution. Both types of speculation have been shown to improve program performance effectively.

Many existing compiler frameworks have already incorporated and used edge/path profiles to support control speculation [Ju et al. 2000, Ishizaki et al. 2002]. Considering the program in Fig. 1(a), the frequency/probability of the execution paths can be collected by edge/path profiling and annotated to the control flow graph. If the branch-taken path (i.e. the condition *c* is true) has a high probability, the compiler can move the load instruction up and execute it speculatively (ld.s) before the branch instruction. A check instruction (chk.s) is inserted at its original location to detect a possible mis-speculation and trigger the recovery code. The ld.s and chk.s are IA64 instructions that support control speculation [Intel 1999]. Similarly, data speculation can be also used to hide memory latency in instruction scheduling [Ju et al. 2000, Ishizaki et al. 2002].



(a) control flow graph of a program

```
ld.s x =[y]
if (c){
    chk.s x, recovery
    next:
    …
}
recovery:
    ld x=[y]
    br next
```

(b) speculative version

Fig. 1. Using control speculation to hide latency.

```
… = *p
*q = ..
… = *p
```

(a) original program

```
r31 = p
ld.a  r32=[r31]
*q = …
ld.c r32=[r31]
…  = r32
```

(b) speculative version

Fig. 2. Using data speculation to eliminate redundancy

However, there has been little work so far to incorporate data speculation into existing compiler frameworks for more aggressive speculative optimizations beyond instruction scheduling. Traditional alias analysis is non-speculative and thus cannot facilitate aggressive speculative optimizations. For example, elimination of redundant loads can sometimes be inhibited by an intervening aliasing store. Considering the program in Fig. 2(a), the traditional redundancy elimination cannot remove the second load of **p* unless the compiler analysis proves that the expressions **p* and **q* *never* access the same location. However, if we know that there is a *small* probability that **p* and **q* will access the same memory location, the second load of **p* can be *speculatively* removed as shown in Fig. 2(b). The first load of **p* is replaced with a speculative load instruction (*ld.a*), and a check load instruction (*ld.c*) is added to replace the second load. If the store of **q* does not access the same location as the load of **p*, the value in register r32 is used directly without re-loading **p*. Otherwise, the value of *p is re-loaded by the ld.c instruction.

In this paper, we address the issues of how to incorporate data speculation into an existing compiler framework and thus enable more aggressive speculative optimizations. We use profiling information and/or simple heuristic rules to supplement traditional non-speculative compile-time analyses. Such information is then incorporated into the static single assignment (SSA) form [Chow et al. 1996].

One important advantage of using data speculation is that it allows *useful* but *imperfect* information, or *aggressive* but *imprecise* heuristics to be effectively used. For example, if we find **p* and **q* are not aliased from the profiling, it does not guarantee that they will not be aliased under other inputs (i.e. *input sensitivity*). We can only assume that they are unlikely to be aliased when we exploit such profiling information in program optimizations. This requires data speculation support in the compiler.

Our extended compiler analysis framework supports both control and data speculation. In this framework, control speculation can be supported by examining the program control structures and estimating the likely execution paths through edge/path profiling

and/or heuristic rules as in [Ball et al. 1993]. Hence, we will focus primarily on the support of data speculation in this paper.

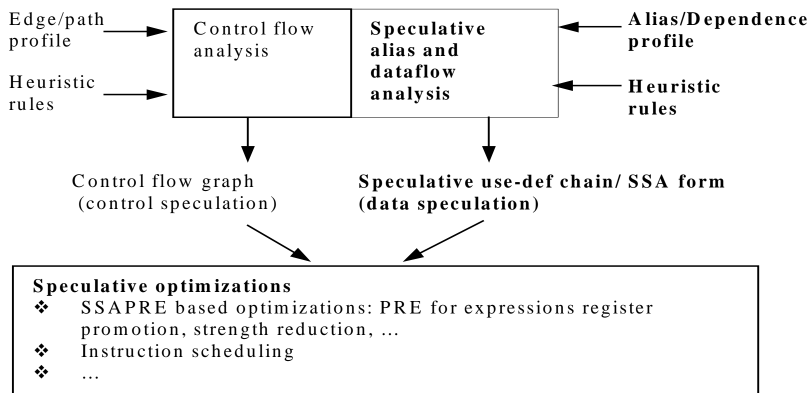


Fig. 3. A framework of speculative analyses and optimizations.

Fig. 3 depicts our framework of speculative analyses and optimizations. This is built on top of the existing SSA framework in the Intel's Open Research Compiler (ORC). While our proposed general framework is not restricted to this particular design, we choose it because it includes a set of compiler optimizations often known as SSAPRE [Kennedy et al. 1999]. Several compiler optimizations, such as redundancy elimination, strength reduction, and register promotion, have been modeled and resolved as partial redundancy elimination (PRE) problems. The existing SSAPRE in ORC already supports control speculation. We extend it by adding data speculation support and related optimizations (see components highlighted in bold in Fig. 3). Our experimental results show that the speculative PRE can be effectively applied to register promotion.

The rest of this paper is organized as follows: We first present an overview of the alias profiling techniques and compiler heuristics in Section 2. In Section 3, we give a survey of related work on data and control speculation, alias analysis and PRE optimization. Next, in Section 4, we present our speculative analysis framework in detail. In Section 5, we propose an algorithm that extends SSAPRE to perform both data and control speculation using the results from speculative analyses. Section 6 presents experimental results on the speculative PRE. Finally, Section 7 summarizes our contributions and concludes.

2. ALIAS PROFILING AND COMPILER HEURISTICS

In order to build an effective framework for speculative analysis and optimizations, we have developed a comprehensive instrumentation-based profiling environment on ORC to generate a set of alias profile [Chen et al. 2002], dependence profile [Chen et al.

2004] and edge/path profile. We also provide a set of compiler heuristics to support speculative analysis. Such profiles and heuristic rules augment compiler analyses with additional information, and can be used to guide effective speculative optimizations.

Fig. 4 illustrates our experimental profiling environment. It includes two major components: an instrumentation tool and a set of library routines that supports profiling. Application programs are first instrumented with calls to the profiling library routines. Alias and/or data dependence information are collected during the execution of the instrumented program with some training inputs.

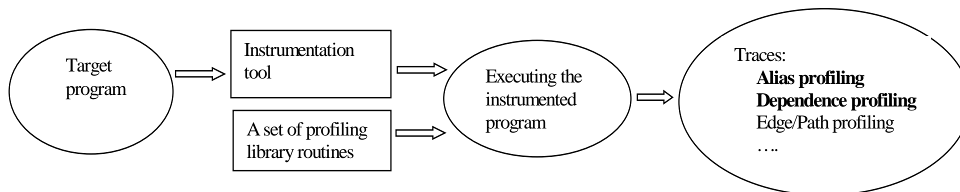


Fig. 4. A typical instrumentation-based profiling process.

The alias profiling collects dynamic points-to sets of every pointer dereference operation. In general, there are two types of points-to targets: the local/global variables defined in the program, and the heap objects allocated at runtime. The local/global variables usually have explicit names given in the program. Whereas, the dynamically allocated heap objects are *anonymous* (i.e. they do not have given names). In order to remedy such a situation, we use both the static line number and the dynamic calling path of the *malloc()* function to assign a name to the allocated objects. In order to control the complexity, we use only the last *n* procedure calls of a calling path in the naming scheme. Different values of *n* will give different levels of granularity to the named memory objects. Such granularity can affect the quality of alias profiling [Chen et al. 2002]. In our experiments, we set *n* to 2, since it gives reasonably good results at a low cost.

A mapping is set up at runtime from the address of a memory object to its name according to the naming scheme. At runtime, the address of an *indirect* memory reference (i.e. a pointer) is known. Targets of a pointer reference are identified by mapping from the address of the reference to the name. The target set thus obtained represents the best result that a naming scheme can be expected to achieve.

However, approximation is introduced into our implementation for the sake of efficiency [Chen et al. 2002]. There are also other limitations in our alias profiling. The alias profiling is a counter part of points-to analysis in a compiler. Both alias profiling and points-to analysis is limited by the finite set of names that are used. For example, array elements or nodes in recursive data structures cannot be distinguished. A program

that implements its own memory allocation functions (such as in *gap* and *perlbmk* in the SPECint 2000 benchmark suite) also poses problems for name assignment. Usually, a compiler has to further resort to other memory disambiguation analyses, such as data dependence test and shape analysis, to handle subscripted expressions and recursive data structures.

To avoid alias profiling overheads, we also implement a set of compiler heuristic rules to provide speculative alias information (see Section 4.2.2). These simple compiler heuristics rules often can produce surprisingly good results for many SPEC2000 benchmark programs (see Section 7.3). However, with such a small set of benchmarks, it is difficult to conclude whether using alias profile or compiler heuristic rules is a more effective way of providing information for our speculative analysis.

3. RELATED WORK

Several recent studies have attempted to use control and data speculation in their program analyses and compiler optimizations such as instruction scheduling, partial redundancy elimination (PRE), register promotion and alias analysis. For example, Ju et al. [Ju et al. 2000] proposed a unified framework to exploit both data and control speculation targeting specifically for *memory latency hiding*. The speculation is exploited by hoisting *load* instructions across potentially aliasing *store* instructions and/or conditional branches, thus allowing memory latency of the load instructions to be hidden. In contrast, our proposed framework is more general in that it can be applied to a much larger set of optimizations.

PRE is a powerful optimization technique first developed by Morel et al [Morel et al 1979]. The technique removes partial redundancy in a program by solving a bi-directional system of data flow equations. Knoop et al. [Knoop et al. 1992] proposed an alternative algorithm, called lazy code motion. It improves on the results of Morel et al by avoiding unnecessary code movement and removing the bi-directional nature of the original PRE data flow equations. The system of equations suggested by Dhamdhere in [Dhamdhere 1991] is weakly bi-directional, and has the same low computational complexity as in uni-directional ones. Chow et al. [Chow et al. 1997] proposed an SSA framework to perform PRE. It used the lazy code motion formulation for *expressions*. Lo et al. [Lo et al. 1998] extended the SSAPRE framework to handle control speculation and register promotion. Bodik et al. [Bodik et al.1998] proposed a PRE algorithm guided by path profile to handle control speculation, and it enables a complete removal of redundancy along more frequently executed paths at the expense of less frequently executed paths. Kennedy et al.

[Kennedy et al. 1998] used the SSAPRE framework to perform strength reduction and linear function test replacement. A recent study by Dulong et al. [Dulong et al. 1999] suggested that PRE could be extended to remove redundancy using both control and data speculation, but no specific schemes were given. In our prior work [Lin et al. 2003], we proposed to use the Advanced Load Address Table (ALAT) [Intel 1999], a hardware feature in IA-64 architecture that supports data speculation, to conduct speculative register promotion. An algorithm for speculative register promotion based on PRE was presented. Later on, we extended our work by introducing speculative SSA form [Lin et al. 2003] to enable effective speculative optimizations.

In this paper, we show that using our proposed framework, SSAPRE can be extended to handle both data and control speculation using alias profiling information and/or simple heuristic rules. Some of our extensions to SSAPRE are similar to the approach used in [Kennedy et al. 1998] for strength reduction.

Most of the proposed alias analysis algorithms [Ghiya et al. 2001, Wilson et al. 1995, Steensgaard 1996, Hind 2001, Diwan et al. 1998] categorize aliases into two classes: (1) *must* alias or *definite* points-to relation, that holds on *all execution paths*, and (2) *may* alias or *possible* points-to relation, that holds on *at least one execution path*. However, it dose not include the information of how *likely* such *may* aliases are real during the program execution. Such information is crucial in data speculative optimizations.

Recently, there have been some studies on speculative alias analysis and probabilistic memory disambiguation. Fernandez et al. [Fernandez et al. 2002] described an approach that used speculative *may* alias information to optimize code. They gave experimental data on the precision and the mis-speculation rates of their speculative analysis results. Ju et al. [Ju. et al. 1999] proposed a method to calculate the alias probability among array references in application programs. Hwang et al. [Hwang et al. 2001] proposed a probabilistic point-to analysis technique to compute the probability of each point-to relation. The memory reference profiling proposed by Wu et al. [Wu et al. 2000] can also be used to calculate the alias probability based on a particular input. However, such profiling can be very expensive since every memory reference needs to be monitored and compared pair-wise. Compared to their approaches, we use an alias profiling scheme [Chen et al. 2002] with a lower cost to estimate the alias probability. In addition, we propose a set of heuristic rules to use in the absence of aliasing profiles.

4. SPECULATIVE ANALALYSIS FRAMEWORK

In this study, we assume the result of the dataflow analysis is in the SSA form [Cytron et al. 1991]. The basic SSA form was originally crafted for scalar variables in sequential programs. It has been extended to cover indirect pointer references [Chow et al. 1996] and arrays [Knobe et al. 1998]. We specify how *likely* an alias relation *may* exist at runtime among a set of scalar variables and indirect references. Such information is then incorporated into an extended SSA form to facilitate data speculation in later program optimizations. The reader is referred to [Chow et al 1996] for a full discussion of the SSA form for indirect memory accesses.

4.1. Basic Concepts

Our speculative SSA form is an extension of the *Hashed SSA* (HSSA) form proposed by Chow et al [Chow et al. 1996]. The traditional SSA form [Cytron et al. 1991] only provides *use-def factored chain* for the scalar variables. In order to accommodate pointers, Chow et al proposed the HSSA form which integrates the alias information directly into the intermediate representation using explicit *may modify operator* (χ) and *may reference operator* (μ). In the HSSA form, *virtual variables* are first created to represent indirect memory references. The rule that governs the assignment of virtual variables is that all indirect memory references that have similar alias behavior in the program are assigned a unique virtual variable. Thus, an alias relation could only exist between *real variables* (i.e. original program variables) and *virtual variables.* In order to characterize the effect of such alias relations, the χ assignment operator and the μ assignment operator are introduced to model the *may modify* and the *may reference* relations, respectively.

In our proposed framework, we further introduce the notion of *likeliness* to such alias relations, and attach a speculation flag to the χ and μ assignment operators according to the following rules:

- Speculative update χ_s: A speculation flag is attached to a χ assignment operator if the χ assignment operator is *highly likely* to be substantiated at runtime.
- Speculative use μ_s: A speculation flag is attached to a μ assignment operator if the μ operator is *highly likely* to be substantiated at runtime.

The compiler can use profiling information and/or heuristic rules to specify the *degree of likeliness* for an alias relation. For example, the compiler can regard an alias relation as *highly likely* if it exists during profiling, and attach speculation flags to the χ and μ assignment operators accordingly. These speculation flags can help to expose opportunities for data speculation.

Fig. 5 shows how to build a use-def chain speculatively by taking such information into consideration. In this example, *v* is a virtual variable to represent **p*, and the numerical subscript of each variable indicates the version number of the variable. Assume variables *a* and *b* are potential aliases of **p*. The fact that the variables *a* and *b* could be potentially updated by the store reference **p* in *s1* is represented by the χ operations on *a* and *b* (i.e. *s2* and *s3*) after the store statement *s1*.

We further assume that according to the profiling information, the indirect memory reference **p* is *highly likely* to be an alias of variable *b*, but not of variable *a*, at runtime. Hence, we could attach a speculation flag for $\chi_s(b_1)$ in *s3* to indicate that the variable *b* is *highly likely* to be updated due to *s1*. Similarly, **p* in *s8* will also be *highly likely* to reference *b*, and we can attach a speculation flag for $\mu_s(b_2)$ in *s7*.

(a) original program	(b) traditional SSA graph	(c) speculative SSA graph
a = …	s0: a_1 = …	s0: a_1 = …
*p = 4	s1: $*p_1 = 4$	s1: $*p_1 = 4$
… = a	s2: $a_2 \leftarrow \chi(a_1)$	s2: $a_2 \leftarrow \chi(a_1)$
a = 4	s3: $b_2 \leftarrow \chi(b_1)$	s3: $b_2 \leftarrow \chi_s(b_1)$
… = *p	s4: $v_2 \leftarrow \chi(v_1)$	s4: $v_2 \leftarrow \chi(v_1)$
	s5: … = a_2	s5: … = a_2
	s6: $a_3 = 4$	s6: $a_3 = 4$
	s7: $\mu(a_3), \mu(b_2), \mu(v_2)$	s7: $\mu(a_3), \mu_s(b_2), \mu(v_2)$
	s8: … = $*p_1$	s8: … = $*p_1$

Fig. 5 Speculative SSA graph

The advantage of having such *likeliness* information is that we could speculatively ignore those updates that do not carry the speculation flag, such as the update to *a* in *s2*, and consider them as *speculative weak updates*. When the update to *a* in *s2* is ignored, the reference of a_2 in *s5* becomes *highly likely* to use the value defined by a_1 in *s0*. Similarly, because **p* is *highly likely* to reference *b* in *s8* (from $\mu_s(b_2)$ in *s7*), we can ignore the use of a_3 and v_3 in *s7*, and conclude that the definition of **p* in *s1* is *highly likely* to reach the use of **p* in *s8*.

From this example, we could see that the traditional SSA form could be easily extended to contain speculation information. The compiler can then use the speculation flags to conduct speculative optimizations later.

4.2. Speculative Alias Analysis and Dataflow Analysis

Fig. 6 shows the main components of the framework of our speculative alias and dataflow analysis.

In this framework, we can use the *equivalence class* based alias analysis proposed by Steensgard [Steensgard 1996] to generate the *alias equivalence classes* for the memory references within a procedure. Each alias class represents a set of *real* program variables. Next, we assign a unique virtual variable for each alias class. We also create the initial μ list and χ list for the indirect memory references and the procedure call statements.

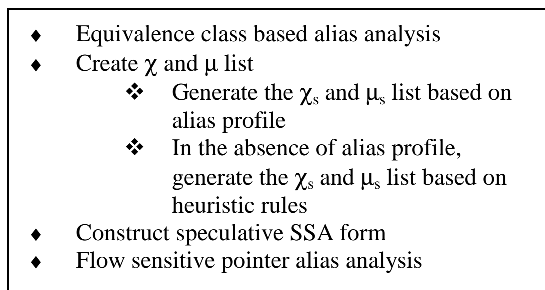


Fig. 6. A Framework for Speculative Alias and Dataflow Analyses.

The rules for the construction of μ and χ lists are as follows: (1) For an indirect memory *store* reference or an indirect memory *load* reference, its corresponding χ list or μ list is initialized with all the variables in its alias class and its virtual variable. (2) For a procedure call statement, the μ list and the χ list represent the *ref* and *mod* information inside the procedure call, respectively.

Using alias profiling information and/or heuristic rules, we construct the χ_s and μ_s lists. In the next step, all program variables and virtual variables are renamed according to the standard SSA algorithm [Cytron et al 1991]. Finally, we perform a flow sensitive pointer analysis using factored use-def chain to refine the μ_s list and the χ_s list. We also update the SSA form if the μ_s and χ_s lists have any change.

4.2.1 Construction of Speculative SSA Form Using Alias Profile

We use the concept of abstract memory locations (LOCs) [Ghiya et al. 2001] to represent the points-to targets in the alias profile. LOCs are storage locations that include local variables, global variables and heap objects. Since heap objects are allocated at runtime, they do not have explicit variable names in the programs[1]. Before profiling, the heap objects are assigned a unique name according to a naming scheme. Different naming schemes may assume different storage granularities [Chen et al. 2002].

For each indirect memory reference, there is a LOC set to represent the collection of memory locations accessed by the reference at runtime. In addition, there are two LOC

[1] For this same reason, the μ and χ lists may not contain a heap object.

sets to represent the side effect information, such as *modified* and *referenced* locations, respectively, at each procedure call site.

The rules of assigning a speculation flag for χ and μ list are as follows:

χ_s: Given an indirect memory *store* reference and its profiled LOC set, add all members in the profiled LOC set to the χ_s list.

μ_s: Given an indirect memory *load* reference and its profiled LOC set, add all members in the profiled LOC set to the μ_s list.

4.2.2 Construction of Speculative SSA Form Using Heuristic Rules

In the absence of alias profile, compiler can also use heuristic rules to assign the speculation flags. We present three possible heuristic rules used in this approach:

1. Any two *indirect* memory references with identical address expressions are assumed *highly likely* to hold the same value.
2. Any two *direct* memory references of the same variable are assumed *highly likely* to hold the same value.
3. Since we do not perform speculative optimization across procedure calls, the side effects of procedure calls obtained from compiler analysis are all assumed *highly likely*. Hence, all χ definitions in the procedure call are changed into χ_s. The μ list of the procedure call remains unchanged.

The above three heuristic rules imply that all updates caused by statements other than call statements between two memory references to the same memory location can be speculatively ignored. Using a trace analysis on the SPEC2000 integer benchmark, we found that these heuristic rules are quite satisfactory with few mis-speculations.

5. SPECULATIVE SSAPRE FRAMEWORK

In this section, we show how to apply the speculative SSA form to speculative optimizations. We use SSAPRE because it includes a set of optimizations that are important to many applications. The set of optimizations in SSAPRE include: *partial redundancy elimination for expressions, register promotion, strength reduction,* and *linear function test replacement* [Kennedy et al. 1999].

5.1. Overview of SSAPRE

Most of the work in PRE is focused on inserting additional computations in the *least likely* execution paths. These additional computations cause *partial* redundant computations in *most likely* execution paths to become *fully* redundant. By eliminating such *fully* redundant computations, we can improve the overall performance.

We assume all expressions are represented as trees with leaves being either constants or SSA renamed variables. We use an extended HSSA form to handle indirect loads. SSAPRE performs PRE on one expression at a time, so it suffices to describe the algorithm with respect to a given expression. In addition, the SSAPRE processes expression trees in a bottom-up order. The SSAPRE framework consists of six separate steps [Kennedy et al 1999]. The first two steps, **ϕ**-*Insertion* and *Rename*, construct an expression SSA form using a temporary variable *h* to represent the *value* of an *expressio*n. In the next two steps, *DownSafety* and *WillBeAvailable*, we select an appropriate set of merge points for *h* that allow computations to be inserted. In the fifth step, *Finalize*, additional computations are inserted in the *least likely* paths, and redundant computations are marked after such additional computations are inserted. The last step, *CodeMotion*, transforms the code and updates the SSA form in the program.

In the standard SSAPRE, control speculation is suppressed in order to ensure the safety of code placement. Control speculation is realized by inserting computations at the incoming paths of a control merge point **ϕ** whose value is not *downsafe* (e.g. its value is not used before it is killed) [Lo et al 1998]. The bold symbol **ϕ** is used to distinguish the merge point in the *expression* SSA form from the merge point in the *original* SSA form, represented as ϕ. Since control speculation may not be beneficial to overall program performance, depending on which execution paths are taken frequently, the edge profile can be used to select the appropriate merge points for insertion.

The *Rename* step plays an important role in facilitating the identification of redundant computations in the later steps. In the original SSAPRE without data speculation, such as the example shown in Fig. 7(a), two occurrences of the *expression a* have the *same* value. Hence, its temporary variable *h* is assigned the *same* version number for those two references. In Fig. 7(a), $[h_1 \leftarrow]$ represents an *action* taken by the compiler on the temporary variable h_1. $\leftarrow$ means a *new* value is to be *loaded* into h_1. Since they have the *same* value, the second occurrence is *redundant*, and can be replaced by a register access.

However, if there is a store **p* that may modify the value of the expression *a*, the second occurrence of *a* is *not* truly redundant and should be assigned a different version number, as shown in Fig. 7(b). In Fig. 7(b), the traditional alias analysis will report that this assignment to **p may* kill the value of the first occurrence of *a*.

Now, as in Fig. 7(c), if the speculative SSA form indicates that the alias relation between the expression of *a* and **p* is *not likely*, we can speculatively assume that the potential update to *a* due to the alias relationship with **p* can be ignored. The second

occurrence of *a* is regarded as *speculatively redundant* to the first one, and a check instruction is inserted to ensure the value of *a* is not changed. This can be done in Intel's IA-64 architecture, for example, by inserting a *ld.c* for the check instruction, and a speculative load instruction *ld.a* for the first load. The register that contains the value in the first occurrence can be used in the second occurrence, instead of reloading it. By *speculatively* ignoring the unlikely update of **p*, we expose *speculative redundancy* between those two occurrences of expression *a*.

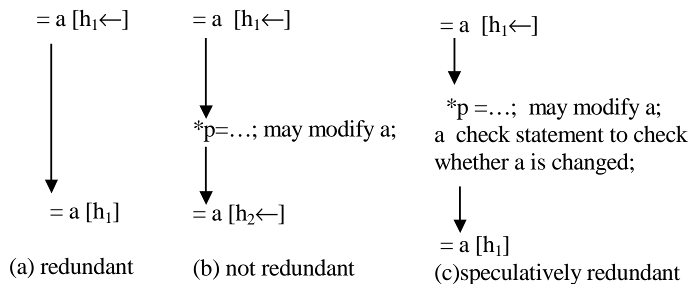


Fig. 7. Types of occurrence relationships (h is temporary variable for a).

Thus, the SSA form built for the variable *a* by the ϕ-Insertion and Rename steps can exhibit more opportunities for redundancy elimination if data speculation is allowed. The generation of check statements such as *ld.c* is performed in *CodeMotion*. The *CodeMotion* step also generates the speculative load flags for those occurrences whose value can reach the check statements along the control flow paths. The changes to support data speculation in the SSAPRE framework are confined to the ϕ-Insertion, Rename and CodeMotion steps.

5.2. ϕ-Insertion Step

One purpose of inserting ϕ's for the temporary variable *h* of an expression is to capture all possible insertion points for the expression. Inserting too few ϕ's will miss some PRE opportunities. On the other hand, inserting too many ϕ's will have an unnecessarily large SSA graph to deal with.

As described in [Kennedy et al. 1999], ϕ's are inserted according to two criteria. First, ϕ's are inserted at the *Iterated Dominance Frontiers* (IDF^+) of each occurrence of an *expression* [Kennedy et al. 1999]. Secondly, a ϕ can be inserted where there is a *ϕ* for a *variable* contained in the *expression*, because it indicates a change of value for the expression that reaches the merge point. The SSAPRE framework performs this type of ϕ insertion in a demand-driven way. An expression at a certain merge point is marked as *not-anticipated* if the value of the expression is either never used before it is killed, or

reaches an exit. A ϕ is inserted at a merge point only if its expression is *partially anticipated* [Kennedy et al. 1999], i.e. the value of the expression is used along one control flow path before it is killed.

For a *not-anticipated* expression at a merge point, if we could recognize that its killing definition is a *speculative weak update*, the expression can become *speculative partially-anticipated*, thus its merge point could be a candidate for inserting computations to allow a *partial redundancy* to become a *speculative full redundancy*.

Fig. 8 gives an example of this situation. In this example, *a* and *b* are *may* alias to **p*. However, *b* is *highly likely* to be an alias of *p, but *a* is *not.* Hence, without any data speculation in Fig. 8(a), the value of a_3 in *s6* cannot reach a_4 in *s13* because of the potential **p* update in *s9*, i.e. a_3 is *not anticipated* at the merge point in *s6*. Hence, the merge point in *s6* is no longer considered as a candidate to insert computations along the incoming paths as shown in Fig. 8(b).

Since *a* is *not likely* to be an alias of **p*, the update of a_4 in *s10* can be *speculatively* ignored, the expression a_3 can now reach a_4 in *s13.* Hence, a_3 in *s6* becomes speculatively *anticipated,* and we could insert a ϕ for temporary variable *h* as shown in Fig. 8(c).

We extended the original algorithm [Kennedy et al. 1999] in the ϕ-Insertion step to handle data speculation. The extended parts that differ from the original algorithm are highlighted in bold.

(a) original program	(b) after traditional ϕ insertion	(c) after enhanced ϕ insertion
s0: … = a_1	s0: … = a_1 [h]	s0: … = a_1 [h]
s1: if (…){	s1: if (…){	s1: if (…){
s2: $*p_1$=…	s2: $*p_1$=…	s2: $*p_1$=…
s3: $a_2 \leftarrow \chi\ (a_1)$	s3: $a_2 \leftarrow \chi\ (a_1)$	s3: $a_2 \leftarrow \chi\ (a_1)$
s4: $b_2 \leftarrow \chi_s\ (b_1)$	s4: $b_2 \leftarrow \chi_s\ (b_1)$	s4: $b_2 \leftarrow \chi_s\ (b_1)$
s5: $v_2 \leftarrow \chi_s\ (v_1)$	s5: $v_2 \leftarrow \chi_s\ (v_1)$	s5: $v_2 \leftarrow \chi_s\ (v_1)$
}	}	}
s6: $a_3 \leftarrow \phi\ (a_1, a_2)$	s6: $a_3 \leftarrow \phi\ (a_1, a_2)$	s6: $h \leftarrow \boldsymbol{\phi}\ (h, h)$
s7: $b_3 \leftarrow \phi\ (b_1, b_2)$	s7: $b_3 \leftarrow \phi\ (b_1, b_2)$	s7: $a_3 \leftarrow \phi\ (a_1, a_2)$
s8: $v_3 \leftarrow \phi\ (v_1, v_2)$	s8: $v_3 \leftarrow \phi\ (v_1, v_2)$	s8: $b_3 \leftarrow \phi\ (b_1, b_2)$
s9: $*p_1$=…	s9: $*p_1$=…	s9: $v_3 \leftarrow \phi\ (v_1, v_2)$
s10: $a_4 \leftarrow \chi\ (a_3)$	s10: $a_4 \leftarrow \chi\ (a_3)$	s10: $*p_1$=…
s11: $b_4 \leftarrow \chi_s\ (b_3)$	s11: $b_4 \leftarrow \chi_s\ (b_3)$	s11: $a_4 \leftarrow \chi\ (a_3)$
s12: $v_4 \leftarrow \chi_s\ (v_3)$	s12: $v_4 \leftarrow \chi_s\ (v_3)$	s12: $b_4 \leftarrow \chi_s\ (b_3)$
s13: … = a_4	s13: … = a_4 [h]	s13: $v_4 \leftarrow \chi_s\ (v_3)$
		s14: … = a_4 [h]

Fig. 8. Enhanced ϕ insertion allows data speculation.

5.3. Rename Step

In the previous subsection, we show how the ϕ-insertion step inserts more ϕ's at the presence of *may-alias* stores, exposing more opportunities for speculative PRE. In

contrast, the Rename step assigns more occurrences of an expression to the *same* version of temporary variable *h* and allows more redundancies to be identified. The enhancement to the Rename step is to deal with *speculative weak update*s and *speculative uses*.

Like traditional renaming algorithms, the renaming step keeps track of the current version of the expression by maintaining a rename stack while conducting a preorder traversal of the dominator tree of the program. Upon encountering an occurrence of a new expression *q*, we trace its use-def chain to determine whether the value of the expression *p* at the top of the rename stack can reach this new occurrence. If so, we assign *q* with the same version as that of the expression *p*. Otherwise, we check whether *q* is speculatively redundant to *p* by ignoring the *speculative weak update* and continue to trace upward along the use-def chain. If we eventually reach the expression *p*, we speculatively assign *q* with the same version as given by the top of the rename stack, and annotate *q* with a speculation flag in order to signal the generation of a check instruction for expression *q* later in the CodeMotion step. If the value of *p* does not reach *q*, we stop and assign a new version for *q*. Finally, we push *q* onto the rename stack and continue the process.

… = a_1 [h_1]	… = a_1 [h_1]
*p_1 = …	*p_1 = …
$v_2 \leftarrow \chi(v_1)$, $a_2 \leftarrow \chi(a_1)$	$v_2 \leftarrow \chi(v_1)$, $a_2 \leftarrow \chi(a_1)$
$b_2 \leftarrow \chi(b_1)$	$b_2 \leftarrow \chi_S(b_1)$
… = a_2 [h_2]	… = a_2 [h_1<speculation>]
(a) traditional renaming	(b) speculative renaming

Fig. 9. Enhanced renaming allows data speculation.

Fig. 9 gives an example that shows the effect of the enhanced renaming. In this example, there are two occurrences of the expression *a* that are represented by the temporary variable *h*. The alias analysis shows that expression **p* and *a may* be aliased. This is represented by the χ operation in the SSA form. These two occurrences of *a* are assigned with different version numbers in the original Rename step. However, in our algorithm, since *p* is not likely to be aliased with *a* (either by alias profile and/or heuristic rules), the χ operation with *a* is *not* marked with χ_s and this update can be ignored in the Rename step. In Fig. 9 (b), the second occurrence of *a* is *speculatively* assigned with the *same* version number as the first one. In order to generate the check instruction in the CodeMotion step, the second occurrence of *a* is annotated with a *speculation flag*. Our algorithm thus successfully recognizes that the first and the second real occurrences of *a*

are of the *same* version by ignoring the *speculative weak update* caused by the indirect reference **p*.

5.4. CodeMotion Step

The CodeMotion step introduces a new temporary variable *t*, which is used to realize the generation of *assignment statements* and *uses* of temporary variable *h* [Kennedy et al. 1999]. This step is responsible for generating *speculative* check statements.

The speculative check statements can only occur at places where the occurrences of an expression are *speculatively partially anticipated.* At the same time, multiple speculative check statements to the same temporary variable should be combined into as few check statements as possible.

The speculative check statements are generated in the main pass of CodeMotion. Starting from an occurrence of *a* with a speculation flag in a use-def chain (shown as "a_2 [h1 <speculation flag>]"in Fig. 10(a)), we reach the first *speculatively weak update* (i.e. "$a_2 \leftarrow \chi(a_1)$" in Fig. 10(a)). A speculative check statement is generated if it has not been generated yet. In our ORC implementation, an *advance load check flag* is attached to the statement first as shown in Fig. 10(b), and the actual speculative check instruction, i.e. ld.c, is generated later in the code generation phase.

… = a_1 [h_1←] *p_1 = … $v_4 \leftarrow \chi\ (v_3)$ $a_2 \leftarrow \chi\ (a_1)$ $b_4 \leftarrow \chi_s\ (b_3)$ … = a_2 [h_1<speculation flag>]	$t_1 = a_1$ (advance load flag) … = t_1 *p_1 = … $v_4 \leftarrow \chi\ (v_3)$ $a_2 \leftarrow \chi\ (a_1)$ $b_4 \leftarrow \chi_s\ (b_3)$ $t_4 = a_2$ (advance load check flag) … = t_4
(a)Before Code Motion	(b) Final Output

Fig. 10. An example of speculative load and check generation.

The occurrences of the temporary variable *h* that are marked with "←" are annotated with an *advanced load flag* (as shown in Fig. 10(b)) if the value of those occurrences can reach their speculative check statements. An actual ld.a instruction will then be generated in the later code generation phase.

In this section, we do not discuss how to eliminate redundant check statements across φ's. Such optimizations can be accomplished using a similar algorithm as in [Kawahito et al. 2000].

6. EXTENSIONS TO SUPPORT MULTI-LEVEL DATA SPECULATION

According to our previous study [Chen et al. 2002], there are many multi-level indirect references (e.g. multi-level field accesses) in the SPEC2000 C programs. Those references exhibit opportunities for speculative register promotion with multi-level data speculation (i.e. cascaded speculation [Ju et al. 2000]) support.

The compiler framework discussed so far focuses on one-level data speculation. This constraint may miss important speculative optimization opportunities. For some indirect reference expressions in the form of ***p*, after we speculatively promote **p* into a register, we may further promote ***p* into a register speculatively. In this section, we enhance our framework to support multi-level data speculation.

One extension is in the speculative Rename step. We ignore the update of speculative check statements generated by the prior level of data speculation when we construct the expression SSA form. For example, consider the following code fragment in Fig. 11(a). After we perform the first-level speculative register promotion, the expression $a \rightarrow b$ can be promoted into a register *r*, and the output is shown in Fig. 11(b). As can be seen, for the expression $r \rightarrow c$, without multi-level speculation support in the Rename step, it cannot be further promoted into a register speculatively because the value of *r* is redefined in the statement s5. With our extension, the definition statement s5 will be ignored in the speculative Rename step. The second occurrence of the expression $r \rightarrow c$ will be assigned the same version number as the first occurrence. Thus, it can be speculatively promoted into a register in the CodeMotion step.

```
a→b→c→d =…
if (…){
        *p = …
        a→b→c→e =…
}
```

(a) original version

```
s1:     r = a→b; (ld.a flag)
s2:     r→c→d =…
s3:     if (…){
s4:             *p = …
s5:             r =a→b; (ld.c flag)
s6:             r→c→e =…
s7:     }
```

(b) the output after one-level data speculation tranformation

Fig. 11. An example to show multi-level data speculation support in the speculative rename step in PRE

Another extension needed to support multi-level speculation in PRE focuses on the CodeMotion step. With multi-level speculation, some of the code may become part of the recovery code, and thus do not exist in the main execution path. In order to assist the generation of recovery code, we introduce a new flag to indicate that the statement should be moved into the recovery code during the later code generation phase.

We use the example in Fig. 12 (a) to show the effectiveness of the multi-level speculation in PRE. We assume the expression *q is aliased with the expression *p, but

not the expression **p. After we perform the first-level data speculation, the value of the first expression *p is kept in the register r as shown in Fig. 12 (b). After we perform the second-level data speculation, the expression *r is promoted into the register s. Since there is no alias relation between expression *r (i.e. **p) and *q, there is no need to use a check for the expression *r. In Fig. 12 (c), it is clear that the statement s6 is unnecessary if the check load in s5 is successful. Therefore, the statement s6 should be moved into the recovery code for the statement s5. We mark the statement s6 with a remove flag to indicate that, during the code generation phase, we need to generate recovery code for the statement s5 and move the statement s6 into the recovery code. The speculative flags guide the later code generation to produce the recovery code as shown in Fig. 12 (d). At present we are working on a new recovery code representation to facilitate the recovery code generation for single and multi-level speculation.

… = **p *q = … (may modify *p) … = **p	r = *p (ld.a flag) …= *r *q = … r = *p (ld.c flag) … = *r	s1: r = *p (ld.a flag) s2: s = *r s3: …= s s4: *q = … s5: r = *p (ld.c flag) s6: s = *r (remove flag) s7: … = s	ld.a r = [p] ld s = [r] …. st [q] =… chk.a r, recovery label: … recovery: ld r = [p] ld s = [r] br label
(a) original program	(b) output after one level data speculation	(c) output after two level data speculation	(d) output of assembly code

Fig. 12. An example to show multi-level data speculation support in speculative code motion step in PRE

7. IMPLEMENTATION AND EXPERIMENTAL RESULTS

We have implemented our speculative PRE algorithm in Intel's Open Research Compiler (ORC) [Ju et al. 2001], version 1.1. The SSAPRE with control speculation is already implemented in ORC. Our implementation extends their work by including data speculative analyses and optimizations. In this section, we study the effectiveness of speculative PRE as applied to register promotion. We measure the effectiveness of our techniques using eight SPEC2000 benchmarks executed with the reference input, while the alias profiling is collected with the train input set. The benchmarks are compiled at the -O3 optimization level (including all intra-procedural optimizations) with type-based alias analysis [Diwan et al. 1998]. The measurements were performed on an HP workstation i2000 equipped with one 733MHz Itanium processor and 2GB of memory running Redhat Linux 7.1. We report on the reduction of dynamic loads, the execution

time speedup *over –O3 performance*, and the data mis-speculation ratio collected by the *pfmon* tool [pfmon 2003].

7.1. The Performance Opportunity in a Sample Procedure

We first use a relatively simple but time critical procedure, *smvp*, in the *equake* program to demonstrate the performance opportunity for the speculative PRE optimization. Procedure *smvp* shown in Fig. 13 takes nearly 60% of the total execution time of *equake*. There are many similar memory references in the inner loop, we show three statements in this example to illustrate the opportunities for speculative register promotion. In this example, the load operations of array ***A* (i.e. *A[][][]*) and ***v* are not promoted to registers because the ***w* references are *possible* aliases as reported by the compiler alias analysis. However, these load operations are speculatively redundant.

```
void smvp(int nodes, double ***A, int *Acol, int *Aindex,
double **v, double **w) {
. . .
  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++;
    }
  }
}
```

Fig. 13. Example code extracted from procedure *smvp*.

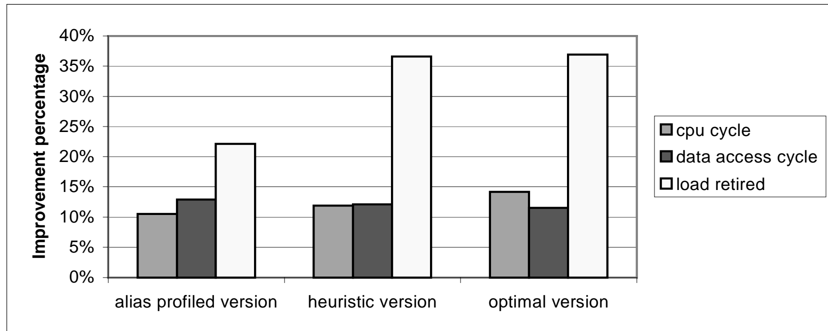


Fig. 14 Performance improvement of different speculative register promotion strategies for procedure *smvp*

Fig. 14 shows the performance improvements using different speculative register promotion strategies over the base version. As can be seen, the speculative register promotion improves the overall performance by more than 10%. In the base version, the ORC compiler adopts a run time disambiguation approach [Nicolau 1989] to keep some references in registers. For example, it reduces the load of *A[Anext][1][1]* after the store

w[col][0] by generating an address comparison between the two references and selecting the proper value to replace the load *A[Anext][1][1]*. However, this approach can only handle simple load-store-load sequences. If there were multiple intervening stores, the number of address comparison instructions need to be generated would increase very quickly. Thus, the load of *A[Anext][2][2]* was not eliminated due to multiple intervening stores like *w[col][0]* and *w[col][1]*.

Data in the first group of bars show the performance of speculative register promotion based on alias profile. Note that the load of ****A* can be speculatively allocated to a register in this version. However, the loads of ***v* cannot be allocated to registers since pointers *v* and *w* point to the same array at the call site and the current alias profile does not distinguish different elements in the same array. As shown in Fig. 14, the speculative register promotion based on heuristic rules reduces more load references and thus improves the performance by more than 10%. The loads for ***v* can be recognized as speculatively loop invariants and hoisted out of the inner loop. As a result, nearly 40% of the load operations in this procedure can be reduced. Finally, we compare the performance of our speculative register promotion with an upper bound where the register allocator aggressively allocates memory references to registers without considering any potential aliases. In Fig. 14, we observe that the optimal case can run 15% faster than the base version. Our speculative register promotion can speed up the code by 12%, approaching the upper bound. In short, our results suggest that a simple local alias analysis based on heuristic rules coupled with speculative register promotion can be practical and cost effective.

7.2. Experimental Data Using Alias Profile

We now examine the effectiveness of speculative register promotion for each benchmark relative to its *base* case, which is highly optimized with –O3 compiler option and *type-based* alias analysis. In general, speculative register promotion shortens the critical paths by promoting the target values of load operations into registers and replacing redundant loads by data speculation checks. Since an integer load has a minimal latency of 2 cycles (L1 Dcache hit on Itanium), and a floating-point load has a minimal latency of 9 cycles (L2 Dcache hit)[2], and a successful check (ld.c or ldfd.c) cost 0 cycle, the critical path could be significantly reduced as long as the speculations are successful.

The first metric in the experiments is the percentage of dynamic loads reduced by speculative register promotion. We also measure the reduction in total CPU cycles and

---

[2] On Itanium, floating point loads fetch data from the L2 data cache directly.

the cycles attributed to data accesses. Fig. 15 shows the results of eight SPEC 2000 programs. The data show that our speculative register promotion based on alias profiles can significantly reduce the number of retired load operations. Among the eight programs, *art*, ammp, *equake, mcf*, and *twolf* have between 5% to 16% reduction in load operations. The reduction of loads in turn reduces data access cycles and CPU cycles. In fact, there exists some overlap between speculative PRE and speculative code scheduling in the reduction of retired loads. In code scheduling, the load is speculative hoisted across the potential aliased store and provide more opportunity for subsequent optimizations, such as peephole to merge adjacent redundant loads. Thus the compiler can achieves the similar effect of speculative PRE, that is, eliminate the possible redundant load or computation. The difference between speculative PRE and speculative code scheduling is speculative PRE can be performed in global manner while speculative code scheduling is limited to local optimization. The ORC compiler performs data speculative instruction scheduling by default. Our experiment data shows the performance gain beyond data speculative instruction scheduling.

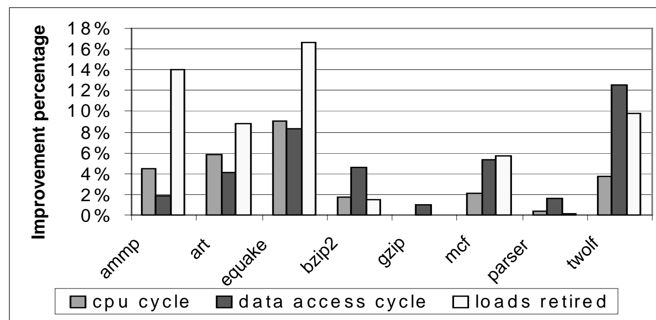


Fig. 15. Performance improvement of speculative register promotion using alias profiles.

As can be observed in Fig. 15, the reduction of loads may not directly translate to execution time improvement. For example, the 6% reduction in loads in *mcf* only achieves a 2% speedup. This is because the eliminated loads are often cache-hit, thus having a smaller impact on performance.

In Fig. 16, we report the percentage of dynamic checks over the total loads retired. It indicates the amount of data speculation opportunities having been exploited in each program. We also report the *mis-speculation ratio* of checks. A high mis-speculation ratio can decrease the benefit of speculative optimization or even degrade performance. In Fig. 16, we observe that the mis-speculation ratio is generally very small. For *gzip*, although the mis-speculation ratio is high, the total number of check instructions is negligible

compared to the total number of load instructions. Therefore, there is no performance impact from this high mis-speculation rate.

Since the profiling is collected with train input set and the programs are executed with reference input set, the low mis-speculation rate also indicate that the alias profiling is quite stable for different input sets. We have experimented with all input sets (test, train and reference) for collecting aliasing profiles and observed very minor performance impact on SPEC2000 benchmarks. This seems to suggest the effectiveness of alias profiling is not very input sensitive.

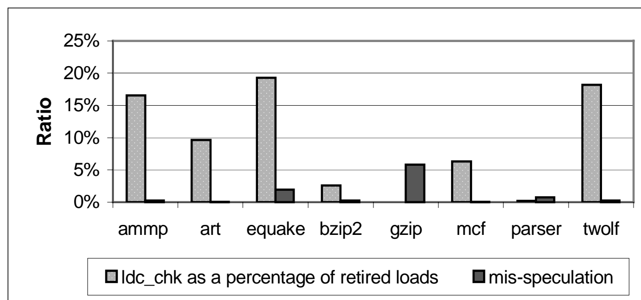


Fig. 16. The mis-speculation ratio in speculative register promotion using alias profile

Speculation has a tendency to extend the lifetime of registers. Register promotion increases the use of registers. The combination of both effects might increase register pressure, and could cause more stack registers to be allocated for the enclosing procedure. More allocated registers may in turn cause memory traffic from more frequent register stack overflow. We have measured the RSE (Register Stack Engine) stall cycles, but have not observed any noticeable increase. Hence, register pressure has not been an issue in our speculative optimizations in these experiments.

7.3. Performance Gain Using Heuristics

In the absence of alias profile, we use heuristic rules to guide speculative register promotion. Using compiler heuristics has several advantages: (1) full coverage: profiling can only cover the part of the program that is reached at runtime; (2) input-insensitive: the heuristics are based only on the structure of a program, and will not be affected by the input set used; (3) low cost: there is no need to collect profiles through instrumentation.

It can be observed that the heuristic rules are quite effective. As shown in Fig. 17, it is comparable to the alias profiling approach for most benchmarks. However, this difference may be attributed to the limitations we have in the current implementation of alias profiling. For example, in order to reduce profiling overhead, we used word (i.e. four bytes) instead of byte as the basic unit for disambiguation. Hence, when sub-word references are made, imprecise but conservative alias information are reported. Another

limitation is that the current alias profiling does not distinguish among field accesses in structures nor for elements in arrays. More accurate profiling information can avoid some of the constraints and yield better performance in speculative optimization with substantially higher profiling overhead. The reader can refer to [Chen et al. 2004] for more details.

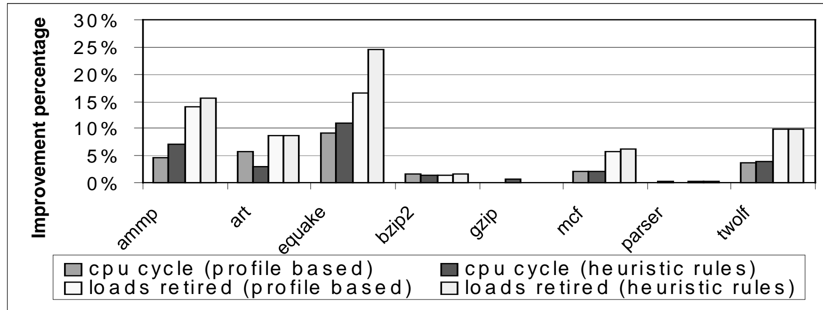


Fig. 17. Performance improvement of speculative register promotion using heuristic rules vs. alias profile.

7.4. Performance with Single Level and Multi-Level Data Speculation

Fig. 19 shows the performance difference of speculative register promotion between single-level speculation and multi-level speculation. Overall, we observe no significant performance change for the benchmarks when the multi-level data speculation is enabled. One reason is that the type-based alias analysis is enabled by default in the base version and is able to disambiguate many memory references. In SPEC2000 C programs, many intervening stores between multiple redundant loads are not of pointer types. Since the type of a multi-level memory reference is usually a pointer type and the type-based alias analysis assumes aliases could only occur between memory references of the same type. Hence, the intervening stores can be ignored and more register promotion opportunities are already exposed. In the code segment shown in Fig. 13, we can see that the expression *A[Anext][1]* can be promoted into a register since the compiler assumes no alias between *w[col][0]* and *A[Anext][1]*. Although many speculative register promotion opportunities have already been realized with type-based alias analysis, it should be noted that the type-based alias analysis is not always *safe*. In contrast, our speculative optimization approach uses runtime checking to guarantee the correctness, so it is a more general approach.

```
*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);
```

Fig. 18. An example of multi-level speculation for expressions including intrinsic calls in *Equake*

Multi-level speculation can improve the performance of some computation expressions common in programs. Consider the code in Fig. 18. The expression *Src.rake* is of the same type with expression **u*. Thus, the typed-based alias analysis will assume they may be aliased with each other. With single-level data speculation, the compiler can only speculatively promote the expression *Src.rake* into registers. Using speculative PRE with multi-level speculation support, the compiler can further promote the expressions *cos(Src.rake)* into a register. Similarly, the expressions *sin(Src.rake)*, *sin(Src.strike)*, *cos(Src.strike)* and *cos(Src.dip)* can also be promoted into registers. This optimization can eliminate six intrinsic calls for the program segment. The performance impact could be substantial for some applications. In Fig. 19, a few cases multi-level data speculation actually hurts performance, particularly in benchmark *gzip*. However, the difference is so small (< 1%) that it is difficult to pin down the real attributors.

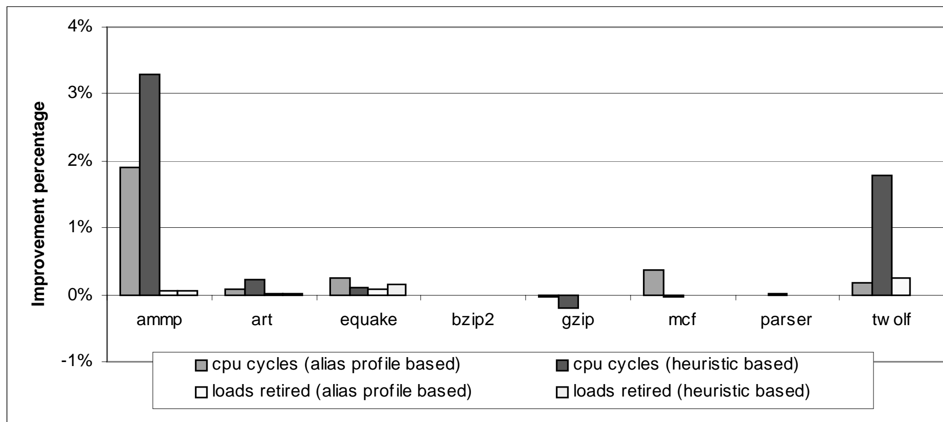


Fig. 19. Performance comparison of speculative register promotion between single level data speculation and multi-level data speculation support

7.5. Potential Load Reduction

We also evaluate the potential of speculative register promotion as opposed to what we have implemented in ORC. We used two methods to estimate the potential load reduction. The first method is simulation-based, similar to the method used in [Bodik 1999]. It measures the number of load reuses in programs. By analyzing the dynamic stream of memory references, we identify all potential speculative reuses available under a given input. The second method uses the existing register promotion algorithm to aggressively allocate memory references to registers assuming no aliases in the program.

In the simulation-based method, we gathered the potential reuses by instrumenting the program after register promotion but before code generation and register allocation. In the simulation, every redundant load is presumed to have its value allocated in a register.

The simulation algorithm assumes a redundant load can reuse the result of another load or itself. The memory references with identical names or syntax trees are classified into the same equivalent classes. A redundant load is detected when two consecutive loads with the same address in an equivalence class load the same value within the same procedure invocation. We track these loads by instrumenting every memory reference and recording its address, value and equivalence class during execution. Fig. 20 shows the numbers of potential load reduction by the simulation-based method and by aggressive register promotion, respectively. We observe that the trend of potential load reduction correlates well with that of the load reduction achieved by our speculative register promotion (c.f. Fig. 15) For example, after seeing the limited potential of *gzip* in Fig. 20, we should not expect a significant performance gain from *gzip*.

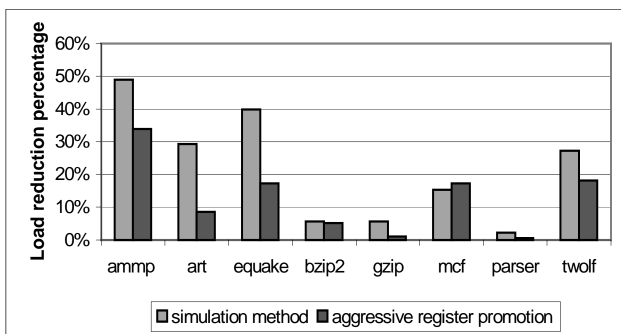


Fig. 20. Potential load reduction.

8. CONCLUSION

In this paper, we propose a compiler framework for speculative analyses and optimizations. Although control speculation has been extensively exploited in existing compiler frameworks, much less work has been done so far to incorporate data speculation into various program optimizations beyond hiding memory latency.

The contributions of this paper are as follows: First, we presented a general compiler framework based on a speculative SSA form to incorporate speculative information for both data and control speculation. Secondly, we demonstrate the use of the speculative analysis in PRE optimizations, which include not only partial redundancy elimination but also register promotion and strength reduction. As a result, many optimizations can be performed aggressively under the proposed framework. Thirdly, this is one of the first

attempts to feed the alias profiling information back into the compiler to guide optimizations. The speculative analysis can be assisted by both alias profile and heuristic rules. Finally, we have implemented the speculative SSA form, and the corresponding speculative analysis, as well as the extension of the SSAPRE framework to take advantage of the speculative analyses. Through the experimental results on speculative register promotion, we have demonstrated the usefulness of this speculative compiler framework and the performance potential of speculative optimizations.

As for future work, we would like to apply data speculation to more optimizations for further performance improvement and to ensure the generality of our framework. We would also like to conduct more empirical studies to further understand the factors that impact the effectiveness of speculative optimizations. One potential performance problem of speculative optimization is that the compiler may oversubscribe the limited entries available in the ALAT. The current size of ALAT is sufficient to accommodate the amount of the current data speculation opportunities in the SPEC programs without exposing any oversubscription problem. However, as more speculative optimizations are developed, the compiler may need to more accurately model the limited ALAT, track the number of concurrently live candidates to ALAT, and properly select candidates to avoid performance loss due to over-subscription.

ACKNOWLGEMENTS

The authors wish to thank Raymond Lo, Shin-Ming Liu (Hewlett-Packard) and Peiyi Tang (University of Arkansas at Little Rock) 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

Ball, T., and Larus , J. 1993. Branch prediction for free. In Proceedings of the ACM SIGPLAN Symposium on Programming Language Design and Implementation, 300-313.

Bodik, R., Gupta, R. and M. Soffa. 1999. Load-reuse analysis: design and evaluation, In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, 64-76.

Bodík, R. , Gupta, R., and Soffa, M. 1998. Complete removal of redundant expressions. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, 1-14.

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 the 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.
Dhamdhere, D. M. 1991. Practical Adaptation of the Global Optimization Algorithm of Morel and Renovise, ACM Trans. on Programming Languages and Systems, 13(2): 291-294.
Diwan, A., McKinley, K., and Moss, J. 1998. Type-based alias analysis, In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, 106-117.
Dulong, C., Krishnaiyer, R., Kulkarni, D., Lavery, D., Li, W., Ng, J. and Sehr, D. 1999. An overview of the Intel IA-64 compiler. Intel Technology Journal.
Fernande, M. and Espasa, R. 2002. Speculative alias analysis for executable code, In Proceedings of International Conference on Parallel Architectures and Compilation Techniques, 222-231.
Ghiya, R., Lavery, D. and Sehr, D. 2001. On the Importance of Points-To Analysis and Other Memory Disambiguation Methods for C Programs. In Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation, 47-58.
Hind, M. 2001. Pointer analysis: Haven't we solved this problem yet? In ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, 54-61.
Hwang, Y.-S., Chen, P.-S., Lee, J.-K. and Ju, R. D.-C. 2001. Probabilistic Points-to Analysis, In Proceeding of the Workshop of Languages and Compilers for Parallel Computing.
Intel Corp. IA-64 Application Developer's Architecture Guide, http://developer.intel.com/design/ia64 /downloads/adag.htm, 1999
Ishizaki, K., Inagaki, T., Komatsu, H. and Nakatani, T. 2002. Eliminating Exception Constraints of Java Programs for IA-64, In Proc. of the Eleventh Int'l Conf. on Parallel Architectures and Compilation Techniques, 259-268.
Ju, R. D.-C., Collard, J. and Oukbir, K. 1999. Probabilistic Memory Disambiguation and its Application to Data Speculation, Computer Architecture News, Vol. 27, No.1.
Ju, R. D.-C., Nomura, K., Mahadevan, U. and Wu, L.-C. 2000. A Unified Compiler Framework for Control and Data Speculation, In Proc. of the Int'l Conf. 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 presented at Micro 34.
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, v.21 n.3, 627-676.
Knobe, K. and Sarkar, V. 1998. Array SSA form and its use in parallelization. In Proceedings of ACM Symposium on Principles of Programming Languages, 107-120.
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. and Yew, P.C. 2003. Speculative Register Promotion Using Advanced Load Address Table (ALAT), In Proceedings of First Annual IEEE/ACM International Symposium on Code Generation and Optimization, 125-134
Lin, J., Chen, T., Hsu, W.C., Yew, P.C., Ju R. D.-C., Ngai, T. F., Chan S. 2003. A Compiler Framework for Speculative Analysis and Optimizations, In Proceedings of ACM SIGPLAN on Programming Language Design and Implementation, 289-299
Lo, R., Chow, F., Kennedy, R., Liu, S. and Tu, P. 1998. Register Promotion by Sparse Partial Redundancy Elimination of Loads and Stores, In Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation, 26-37.
Morel, E. and Renvoise, C. 1979. Global optimization by suppression of partial redundancies. Communications of the ACM, 22(2): 96-103.
Nicolau, A. 1989. Run-time disambiguation: Coping with statically unpredictable dependencies. IEEE Transactions on Computers, 38(5):663-678.
Pfmon 2003: ftp://ftp.hpl.hp.com/pub/linux-ia64/pfmon-1.1-0.ia64.rpm
Steensgaard, B. 1996. Points-to analysis in almost linear time. In Proceedings of ACM Symposium on Principles of Programming Languages, 32-41.
Wilson, R.P. and Lam, M.S. 1995. Efficient context-sensitive pointer analysis for C program. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, 1-12.
Wu, Y. and Lee, Y. 2000. Accurate Invalidation Profiling for Effective Data Speculation on EPIC processors, In 13th International Conference on Parallel and Distributed Computing Systems.