A Compiler Framework for Speculative Analysis and Optimizations

Jin Lin Tong Chen Wei-Chung Hsu Roy Dz-Ching Ju Tin-Fook Ngai
Pen-Chung Yew Sun Chan

Department of Computer Science and Engineering
University of Minnesota
Minneapolis, MN 55455 U.S.A

{jin, tchen, hsu, yew}@cs.umn.edu

Microprocessor Research Lab.
Intel Corporation
Santa Clara, CA 95052 U.S.A

{roy.ju, tin-fook.ngai, sun.c.chan}@intel.com

ABSTRACT

Speculative execution, such as control speculation and 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 program analysis frameworks to be easily extended to support both control and data speculation. Such a general framework is very useful for EPIC architectures that provide checking (such as *advanced load address table* (ALAT) [10]) on data speculation to guarantee the correctness of program execution. We use SSAPRE [21] as one example to illustrate how to incorporate data speculation in those important compiler optimizations such as partial redundancy elimination (PRE), register promotion, strength reduction and linear function test replacement. 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 and how data speculation benefits partial redundancy elimination.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors – *compiler, optimization.*

General Terms

Algorithms, Performance, Design, Experimentation.

Keywords

Speculative SSA form, speculative weak update, data speculation, partial redundancy elimination, register promotion.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PLDI '03, June 9-11, 2003, San Diego, California, USA.
Copyright 2003 ACM 1-58113-662-5/03/0006…$5.00.

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 are effective techniques to improve program performance.

Many existing compiler analysis frameworks have already incorporated and used edge/path information to support control speculation. Considering the program in Figure 1, the frequency/probability of the execution paths can be collected by 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 up and execute it speculatively (ld.s) before the branch instruction. A check instruction (chk.s) is inserted at its home location to catch and recover from any invalid speculation. The ld.s and chk.s are IA64 instructions that support control speculation [10]. Since the execution of the speculative load may overlap with the execution of other instructions, the critical path can be shortened along the speculated path.

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

(a) original program (b) speculative version

Figure 1. Using control speculation to hide memory latency.

However, so far there has been little work on how to incorporate information for data speculation into existing compiler analysis frameworks to help 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 Figure 2(a), the traditional redundancy elimination cannot remove the second load **p* unless the compiler analysis proves that the expressions **p* and **q* do not access the same location. However, through profiling or simple

heuristic rules, 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 Figure 2(b). The first load **p* is replaced with a speculative load instruction (*ld.a*), and a check load instruction (*ld.c*) is added to replace the second load instruction. If the store of **q* does not access the same location as the load **p*, the value in register r32 is used directly without re-loading **p*.

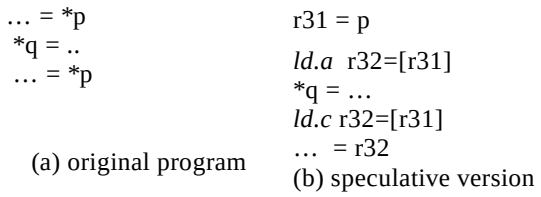

Figure 2. Redundancy elimination using data speculation.

In this paper, we address the issues of how to incorporate information for data speculation into an existing compiler analysis framework and thus enable aggressive speculative optimizations. We use profiling information and/or simple heuristic rules to supplement traditional non-speculative compile-time analysis. Such information is then incorporated into the SSA form.

One important advantage of using data speculation is that it allows us to use useful but imperfect information or to apply aggressive but uncertain heuristic rules. For example, if we find **p* and **q* are not aliases in the current profiling, it does not guarantee that they are not aliases under different program inputs (i.e. *input sensitivity*). We can only assume *speculatively* that they are not aliases when we exploit such profiling information in program optimizations. This requires data speculation support.

Our extended compiler analysis framework supports both control and data speculation. Like traditional compiler analysis, control speculation is supported by examining program control structures and estimating likely execution paths through edge/path profiling and/or heuristic rules [1]. In this paper, we will focus on the data speculation support in the extended framework

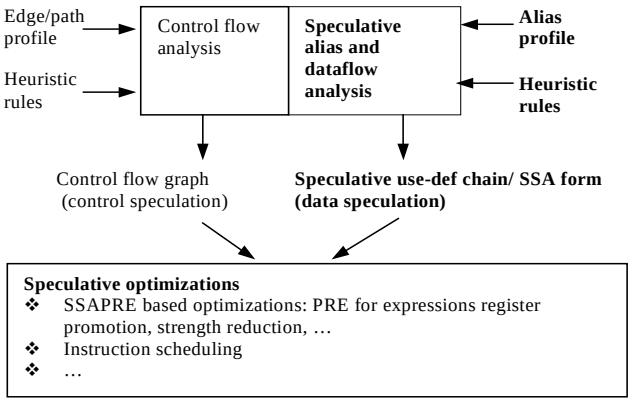

Figure 3. A framework of speculative analyses and optimizations.

Figure 3 depicts our framework of speculative analysis and optimization. This is built on top of the existing SSA framework in the ORC compiler. While the general framework we proposed is not restricted to this particular design, we choose it to exemplify our framework because it includes a set of compiler optimizations often known as SSAPRE [21]. Many optimizations problems, such as redundancy elimination, strength reduction, and register promotion, have been modeled and resolved as PRE problems. The existing SSAPRE in ORC already supports control speculation. We extend it by adding data speculation support and speculative optimizations (see components highlighted in bold in Figure 3). In our experimental results, we study the effectiveness of speculative PRE as applied to register promotion.

The rest of this paper is organized as follows: We first give a survey on related work on data and control speculation, alias analysis and PRE optimization in section 2. Then, in section 3, we present our speculative analysis framework in detail. Next, in section 4, we propose an algorithm that extends SSAPRE to perform both data and control speculation using the speculative analysis results. Section 5 presents some experimental results on the speculative PRE. Finally, section 6 concludes this paper by summarizing our contributions.

2. RELATED WORK

Several recent studies have tried to use control and data speculation to help program analysis and compiler optimizations such as instruction scheduling, PRE, register promotion and alias analysis.

For example, Ju et al. [17] 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 or conditional branches, thus allow memory latency of the load instruction to be hidden. In contrast, our proposed framework is a general framework that can exploit a larger set of optimizations, such as those in PRE and instruction scheduling, by utilizing general profiling information or incorporating general heuristic rules.

PRE is a powerful optimization technique first developed by Morel et al [26]. The technique removes partial redundancy in programs by solving a bi-directional system of data flow equations. Knoop et al. [23] proposed an alternative PRE algorithm called lazy code motion that improves on Morel et al 's results 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 [8] is weak bi-directional and have the same low computational complexity as uni-directional ones. Chow et al. [6, 21] was the first one to propose an SSA framework to perform PRE, which used the lazy code motion formulation for *expressions*. Lo et al. [25] extended the SSAPRE framework to handle control speculation and register promotion. Bodik et al. [3] proposed a path profile guided PRE algorithm to handle control speculation so that it enables the complete removal of redundancy along more frequent paths at the expense of additional computation along less frequently executed paths. Kenendy et al. [20] used the SSAPRE framework to perform strength reduction and linear function test replacement. A recent study by Dulong et al. [11] suggested that PRE can be extended to remove redundancy using both control and data speculation, but no systematic design were given. In our prior work [24], we studied the use of the Advanced Load Address

Table (ALAT), a hardware feature to support data speculation defined in the IA-64 architecture, for speculative register promotion. An algorithm for speculative register promotion based on PRE was presented.

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 handle data speculation in SSAPRE are similar to the approach used in [20] for strength reduction. The speculative weak update concept described in this paper corresponds to the *injuring definition* and the generation of speculative check instructions corresponds to the *repair code* in [20]. On the other hand, our work builds a unified speculation framework to allow SSAPRE be easily extended to incorporate both data and control speculation.

Most of the proposed alias analysis algorithms [13, 29, 28, 14, 9] categorize aliases into two classes: *must* alias or *definite* points-to relation, which holds for all execution paths, and *may* aliases or *possible* points-to relation, which may hold for at least one execution path. However, they did not include the information of how *likely* such *may* aliases may occur during the program execution. Such information is very crucial in data speculative optimizations.

Recently, there has been some studies on speculative alias analysis and probabilistic memory disambiguation. Fernandez [12] described some approaches that use speculative *may* alias information to optimize code. They gave some experimental data on the precision and the mis-speculation rates in their speculative analysis results. Ju. et al.[16] proposed a method to calculate the alias probability among array references in application programs. Hwang et al. [15] proposed a probabilistic point-to analysis technique to compute the probability of each point-to relation. It could be used to guide the calculation of alias probability among pointer references. The memory reference profiling proposed by Wu et al [30] 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. It could slow down the program execution by order of magnitude during the profiling phase. Compared to their approaches, we use a lower cost alias profiling scheme to estimate the alias probability. In addition, when alias profiling is unavailable, we use a set of heuristic rules to quickly approximate alias probabilities in common cases.

3. SPECULATIVE ANALALYSIS FRAMEWORK

In this study, we assume the result of the dataflow analysis is in the SSA form. In SSA, each definition of a variable is given a unique version number. Different versions of the same variable can be regarded as different program variables. Each use of the variable can only refer to a single reaching definition of some version of that variable. When several definitions of a variable, a_1, a_2, …,a_{n-1}, reach a merge point in the control flow graph, a ϕ function is inserted to merge them into the definition of a new version a_n, i.e. $a_n \leftarrow \phi(a_1, a_2, \ldots, a_{n-1})$. Thus, the semantic of *single assignment* is preserved.

The introduction of a new version as the result of a ϕ function can factor the set of use-def edges over merge nodes, and thus reduce the number of use-def edges needed. The basic SSA form was originally crafted for scalar variables in sequential programs. Recently, it has been extended to cover indirect pointer references [5] and arrays [22].

In this paper, we further 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 [5, 19] for a full discussion on the SSA form for indirect memory accesses.

3.1 Basic Concepts

Our speculative SSA form is an extension of the HSSA form proposed by Chow et al [5, 19]. The traditional SSA form [7] 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 behaviors 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. It indicates that this update is *highly likely* and can't be ignored.

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. It indicates that the variable in the μ assignment operator is *highly likely* to be referenced during the program execution.

The compiler can use the profiling information and/or some 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.

Example 1 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 **p* store reference in *s1* is represented by the χ operations on *a* and *b* after the store statement.

We further assume that according to the profiling information, the indirect memory reference **p* is *highly likely* to be an alias of the

variable *b*, but not of the variable *a*, at runtime. Hence, we could attach a speculation flag for $\chi_s(b_1)$ in *s3* because the update to *b* caused by the potential store **p* is also *highly likely* to be executed. 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*.

Example 1

s0:	$a_1 = \ldots$	s0:	$a_1 = \ldots$
s1:	$*p_1 = 4$	s1:	$*p_1 = 4$
s2:	$a_2 \leftarrow \chi(a_1)$	s2:	$a_2 \leftarrow \chi(a_1)$
s3:	$b_2 \leftarrow \chi(b_1)$	s3:	$b_2 \leftarrow \chi_s(b_1)$
s4:	$v_2 \leftarrow \chi(v_1)$	s4:	$v_2 \leftarrow \chi(v_1)$
s5:	$\ldots = a_2$	s5:	$\ldots = a_2$
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:	$\ldots = *p_1$	s8:	$\ldots = *p_1$
	(a) traditional SSA graph		(b) 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 speculative SSA form could contain both traditional compiler analysis information and speculation information. The compiler can use the speculation flags to conduct speculative optimizations.

3.2 Speculative Alias Analysis and Dataflow Analysis

- Equivalence class based alias analysis
- Create χ and μ list
  - Generate the χ_s and μ_s list based on alias profile
  - In the absence of alias profile, generate the χ_s and μ_s list based on heuristic rules
- Construct speculative SSA form
- Flow sensitive pointer alias analysis

Figure 4. A Framework of Speculative Alias and Dataflow Analysis.

Figure 4 shows a basic framework of the alias analysis and the dataflow analysis with the proposed extension to incorporate speculation flags to the χ and μ assignment operators using profiling information and/or heuristic rules.

In this framework, we can use the *equivalence class* based alias analysis proposed by Steensgard [28] 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.

The rules of 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 of 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 [7]. 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.

In the following sections we give more detailed description on how to construct speculative SSA form using alias profile and heuristic rules.

3.2.1 Construction of Speculative SSA Form Using Alias Profile

We use the concept of abstract memory locations (LOCs) [13] 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 [4].

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 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, if any of the member in its profiled LOC set is not in its χ list, add the member to the χ list using the speculation update χ_s. If the member is in its χ list, then a speculation flag is attached to its χ operator (thus becoming a speculative update χ_s).

μ_s:

Given an indirect memory *load* reference and its profiled LOC set, if any of the member in its profiled LOC set is not in its μ list, add the member to the μ list using the speculative use μ_s. If the member is in its μ list, then a speculation flag is attached to its μ operator (thus becoming a speculative use μ_s).

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

3.2.2 Construction of Speculative SSA Form Using Heuristic Rules

In the absence of alias profile, compiler can also use some heuristic rules to assign the speculation flags. The heuristic rules discussed here are based on the pattern matching of syntax tree. We present three possible heuristic rules used in this approach:

1. The two *indirect* memory references with an identical address expression are assumed *highly likely* to hold the same value.
2. The 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 with the same syntax tree can be speculatively ignored. Using a trace analysis on SPEC2000 integer benchmark, we found that these three heuristic rules are quite satisfactory with surprisingly few mis-speculations.

4. SPECULATIVE SSAPRE FRAMEWORK

In this section, we show how to apply the speculative SSA form for speculative optimizations. We use SSAPRE [21] because it includes a set of optimizations that are important in most compilers. The set of optimizations in SSAPRE include: *partial redundancy elimination for expressions, register promotion, strength reduction and linear function test replacement.* We first give a quick overview of the SSAPRE framework. Then, we present an extension to incorporate both data and control speculation.

4.1 Overview

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 then improve the overall performance.

We assume all expressions are represented as trees with leaves being either constants or SSA renamed variables. For indirect loads, the indirect variables have to be in SSA form in order for SSAPRE to handle them. Using an extended HSSA form presented in [5], it can uniformly handle indirect loads together with other variables in the program.

SSAPRE performs PRE one expression at a time, so it suffices to describe the algorithm with respect to a given expression. In addition, the SSAPRE processes the operations in an expression tree using a bottom-up order. The SSAPRE framework consists of six separate steps [21]. 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 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) [25]. The symbol ϕ is used to distinguish the merge point in the *expression* SSA form which is different from the merge point ϕ in the *original* SSA form. Since control speculation may or may not be beneficial to overall program performance, depending on which execution paths are taken frequently, the edge profile of the program 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 Figure 5(a), two occurrences of an *expression a* have the *same* value, hence, its temporary variable *h* are assigned the *same* version number for those two references. Since they have the *same* value, the second occurrence is *redundant* to the first one, thus, the second load can be replaced with a register access. "$h_1\leftarrow$" in Figure 5(a) means a value is *to be stored* into h_1.

However, if there is a store **p* that may modify the value of the expression *a*, the second occurrence of *a* is *not* redundant and should be assigned a different version number, as shown in Figure 5(b). In Figure 5(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 Figure 5(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 to **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 check whether the value of *a* is changed before the second occurrence of *a* (This can be done, for example, by inserting a *ld.c* instruction on the IA-64 architecture [10]). The register that contains the value in the first occurrence can be used in the second occurrence, instead of reloading it. By *speculatively* ignoring those updates, we expose *speculative redundancy* between those two occurrences of the expression *a*.

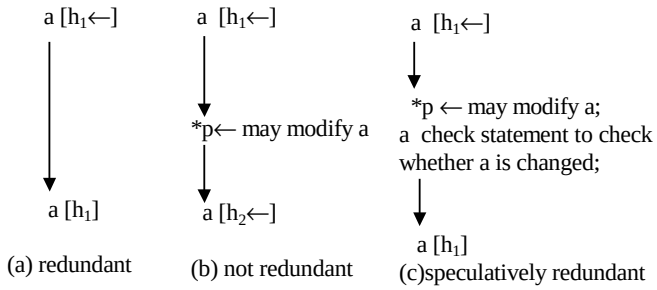

Figure 5. Types of occurrence relationships (h is temporary variable for a).

Thus, the SSA form built for the variable *a* by the ϕ-Insertion and Rename steps can exhibit more opportunities for redundancy

elimination if it is enhanced to allow data speculation. The generation of check statements 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 SSAPRE framework are confined to **ϕ**-Insertion, Rename and CodeMotion steps.

We now give more detailed description on the extension to incorporate data speculation in the SSAPRE framework. The reader is referred to [21] for a full description of the foundation of SSAPRE.

4.2 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 be dealt with.

As described in [21,6], **ϕ**'s are inserted according two criteria. First, **ϕ**'s are inserted at the *Iterated Dominance Frontiers* (DF^+) of each occurrence of an *expression* [21]. 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 defined as *not anticipated* if the value of the expression is 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* [21], 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 *partially-anticipated speculatively*, thus its merge point could potentially be a candidate for inserting computations to allow a *partial redundancy* to become a *speculative full redundancy*.

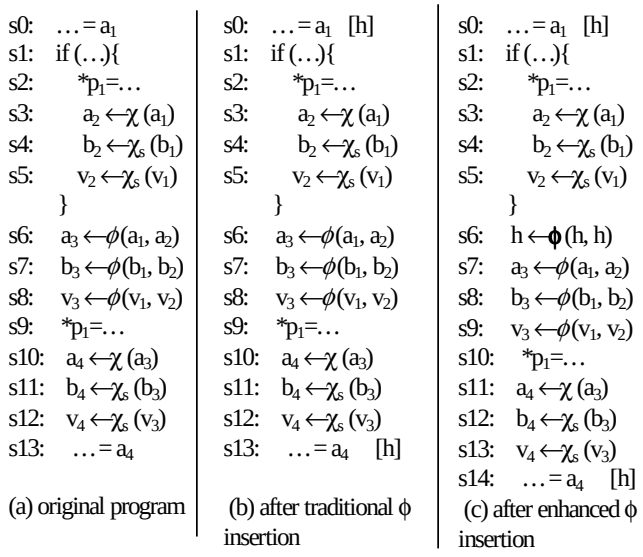

Figure 6. Enhanced ϕ insertion allows data speculation.

Figure 6 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 likely* to be an alias of *p. Hence, without any data speculation in Figure 6(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 Figure 6(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 Figure 6(c).

Appendix A gives the extended version of the **ϕ**-Insertion step that handles data speculation. The parts that differ form the original algorithm [21] are highlighted in bold.

4.3 Rename Step

In the previous subsection, we show how the **ϕ**-insertion step inserts more **ϕ**'s at the presence of *may-alias* stores, creating more opportunities for inserting more computations. 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 rename stack while conducting a preorder traversal of the dominator tree of the program. Upon encountering a new expression occurrence *q*, we trace the use-def chain to determine whether the value of the expression *p* on top of 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 speculative redundant to *p* by ignoring the *speculative weak update* and continuing tracing 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 the rename stack and annotate *q* with a speculation flag in order to enforce the generation of check instruction for expression *q* later in the code motion step. If the value of *p* cannot reach *q*, we stop and assign a new version for *q*. Finally, we push *q* onto the rename stack and proceed.

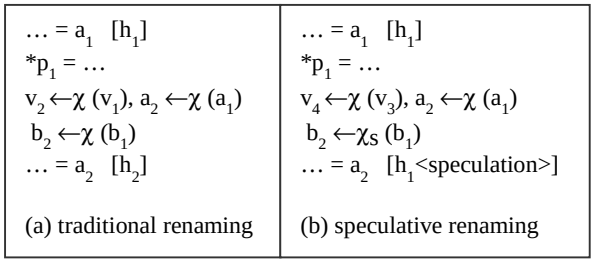

Figure 7. Enhanced renaming allows data speculation.

Figure 7 gives an example that shows the effect of 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 aliases. Variable *a may be* updated after the store of **p*, and 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, if *p* does not point to *a* (either by alias profile and/or heuristic rules), the χ operation with *a* is *not* marked with χ_s, so this update can be ignored in the Rename step. In Figure 7 (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*. So our algorithm successfully recognizes that the first and the second real occurrences of *a* are in the *same* version by ignoring the *speculative weak update* caused by the indirect reference **p*.

4.4 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* [21]. With data speculation, this step is also responsible for generating *speculative* check statements.

The speculative check statements can only occur at places where the occurrences of an expression are *partially anticipated speculatively*. 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 *a* with a speculation flag in a use-def chain (shown as "a_2 [h1 <speculation flag>]"in Figure 8(a)), we reach the first *speculatively weak update* (i.e. "$a_2 \leftarrow\chi(a_1)$" in Figure 8(a)). A speculative check statement is generated if it has not been generated yet. In our ORC implementation, actually an *advance load check flag* is attached to the statement first as shown in Figure 8(b), and the real speculative check instruction, i.e. ld.c, is generated later in the code generation phase.

The occurrences of the temporary variable *h* that are marked with "←" are annotated with an *advanced load flag* (as shown in Figure 8(b)) if the value of those occurrences can reach their speculative check statements. An actual ld.a instruction will be then generated in the later code generation phase.

In Appendix B we give the extended version of the CodeMotion step that handles data speculation.

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

Figure 8. An example of speculative load and check generation.

5. EXPERIMENTAL RESULTS

We have implemented our speculative PRE algorithm in the Open Research Compiler (ORC) [18], version 1.1. The SSAPRE with control speculation is already implemented in ORC. Our implementation extends their work by including data speculative analysis and optimizations. In this section, we study the effectiveness of speculative PRE as applied to register promotion. Speculation in software pipelining is not included in the current implementation. We measure the effectiveness of our techniques using eight SPEC2000 benchmarks executed with the reference inputs. The benchmarks are compiled at the –O3 optimization level with type-based alias analysis [9]. 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 [27].

5.1 The Performance Opportunity Exhibited in a Procedure

We first use a relatively simple but time critical procedure, *smvp*, in the *equake* program to demonstrate the performance opportunity of our implemented speculative PRE optimization. Procedure *smvp* shown in Figure 9 takes nearly 60% of the total execution time of *equake*. There are many memory references of similar patterns in the inner loop, and we show three statements in this example to illustrate the speculative register promotion opportunities. In this example, the load operations of array ****A* (i.e. *A[][][]*) and ***v* are not promoted to registers because the ***w* references are *possibly* aliased with them as reported in the compiler alias analysis. However, these load operations can be speculatively promoted into registers.

```
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++;
    }
  }
}
```

Figure 9. Example code extracted from procedure *smvp*.

According to our alias profile feedback, these potential aliasing never actually occur at runtime. Hence, it would be profitable to speculatively promote ****A* and ***v* to registers. Furthermore, all ***v* references can be treated as loop invariants and speculatively hoisted out of the inner loop. As a result, 39.8% of all load

operations in this procedure can be replaced by check instructions.

After our speculative register promotion transformation, procedure *smvp* is 6% faster than the base version. As a reference point, a manually tuned *smvp*, which allocates the aforementioned candidates to registers without generating any check instructions[2], can be 14% faster than the base version. This indicates that we should be able to gain a lot more from procedure *smvp*. After inspecting the generated Itanium code sequence, we learn that our transformation replaces regular floating point loads by check instructions (ldfd.c), and the current instruction scheduler in ORC does not effectively schedule floating point load check instructions. Some tuning in the instruction scheduler could significantly boost the performance of the transformed *smvp*.

5.2 Experimental Data for Eight SPEC2000 Benchmarks

We now examine the effectiveness of speculative register promotion for each benchmark relative to its *base* case, which is already highly optimized with the –O3 compiler option and *type-based* alias analysis.

In general, speculative register promotion shortens the critical paths by promoting the 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)[3], and a successful check (ld.c or ldfd.c) cost 0 cycles, the critical path could be significantly reduced as long as the speculations are successful.

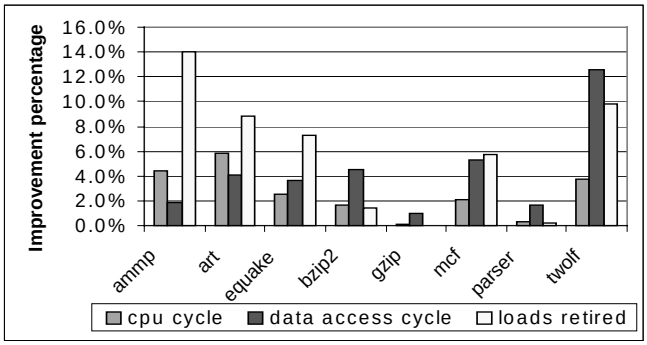

Figure 10. Performance improvement of speculative register promotion using alias profiles.

The first metric is the percentage of load operations reduced by speculative register promotion at runtime. We also measure the reduction in total CPU cycles and the cycles attributed to data access. Figure 10 shows results from eight SPEC 2000 programs from our implementation. The data show that our speculative register promotion based on alias profiles can significantly reduces retired load operations. Among the eight programs, *art*, ammp, *equake*, mcf, and *twolf* have between 5% to 14% reduction on load operations. The reduction of loads in turn reduces data access cycles and CPU cycles.

As can be observed from Figure 10, the reduction of loads may not directly translate into execution time improvement. For example, 6% of loads reduction in *mcf* only achieves 2% of execution time speedup. This is because the reduced loads are often cache-hit operations, thus having a smaller impact on performance for programs suffering from frequent data cache misses.

In Figure 11, we report the percentage of dynamic check loads over the total loads retired, which indicates the amount of data speculation opportunities having been exploited in each program. We also report the percentage of load checks that failed during runtime, and this metric is called the *mis-speculation ratio*. A high mis-speculation ratio can decrease the benefit of speculative optimization or even degrade performance. In Figure 11, we observe that the mis-speculation ratio is generally very small. For *gzip*, although the mis-speculation ratio is almost 6%, the total number of check instructions is nearly negligible compared to the total number of load instructions. Therefore, there is little performance impact from the high mis-speculation ratio.

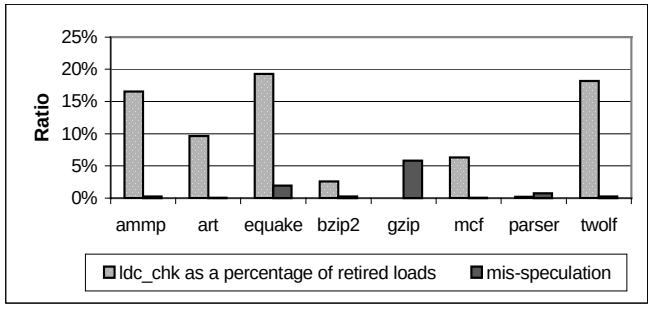

Figure 11. The mis-speculation ratio in speculative register promotion.

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, which could cause more stack registers to be allocated for the enclosing procedure. More allocated registers may in turn cause memory traffic from register stack overflows. We have measured the RSE (Register Stack Engine) stall cycles, but have not observed any notable increase. Hence, register pressure has not been an issue from our speculative optimizations in these experiments.

In the absence of alias profile, we apply heuristic rules in our speculative analysis framework and use this information for speculative register promotion. We have performed similar experiments to evaluate the heuristic version. We found that the performance of the heuristic version is *comparable* to that of the profile-based version.

5.3 Potential Load Reduction

We also evaluate the potential of speculative register promotion by comparing the number of load reduction currently exploited in our implementation to the number of speculatively redundant loads visible at runtime. We used two methods to estimate the

[2] Since there is no aliasing observed during run-time, the optimistic code runs correctly when the same input set is used.

[3] On Itanium, floating point loads fetch data from the L2 data cache

potential load reduction. The first method is simulation-based, similar to the method used in [2]. It can measure the amount of load reuses in programs. By analyzing the dynamic stream of memory references, we can identify all potential speculative reuses available under a given input. The second method uses the existing register promotion algorithm, but aggressively allocates memory references into registers without considering any potential alias.

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

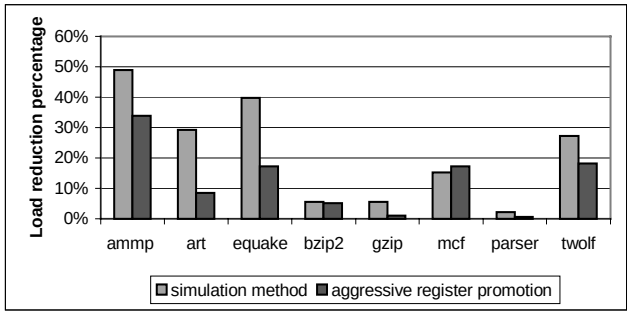

Figure 12. Potential load reduction.

The simulation algorithm assumes a redundant load can reuse a result of another load or itself. These loads have identical names (if they are scalars) or identical syntax tree structures. The memory references with identical names or syntax trees are classified into the same equivalent classes. redundancies are detected by tracking the access behavior of each static memory reference. 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. Figure 12 shows the numbers of potential load reduction by the simulation-based method and 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. Figure 10.) For example, after seeing the limited potential of *gzip* in Figure 12, we may not expect a significant performance gain from speculative register promotion.

6. CONCLUSIONS

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

The contributions of this paper are as follows: Firstly, we presented a general compiler analysis 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 compiler analysis 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 use the speculative analysis results. Through the experimental results on speculative register promotion, we have demonstrated the usefulness of this speculative compiler framework and promising performance potential of speculative optimizations.

As for future work, we would like to enable more optimizations under the speculative SSA form to ensure the generality of the framework and exploit additional performance opportunities. We would also like to conduct more empirical studies to further understand the factors that impact the effectiveness of speculative optimizations.

7. ACKNOWLGEMENT

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.

8. REFERENCES

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

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

[3] R. Bodík, R. Gupta, and M. Soffa. Complete removal of redundant expressions. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, pages 1-14, Montreal, Canada, 17-19 June 1998.

[4] T. Chen, J. Lin, W. Hsu, P.C. Yew. An Empirical Study on the Granularity of Pointer Analysis in C Programs, In 15th Workshop on Languages and Compilers for Parallel Computing, pages 151-160, College Park, Maryland, July 2002.

[5] F. Chow, S. Chan, S. Liu, R. Lo, and M. Streich. Effective representation of aliases and indirect memory operations in SSA form. In Proceedings of the Sixth International Conference on Compiler Construction, pages 253--267, April 1996.

[6] F. Chow, S. Chan, R. Kennedy, S. Liu, R. Lo, and P. Tu. A new algorithm for partial redundancy elimination based on SSA form. In Proceedings of the ACM SICPLAN Conference on Programming Language Design and

Implementation, pages 273-286, Las Vegas, Nevada, May 1997.

[7] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. ACM Transactions on Programming Languages and Systems, 13(4): 451--490, 1991.

[8] D. M. Dhamdhere. Practical Adaptation of the Global Optimization Algorithm of Morel and Renovise, ACM Trans. on Programming Languages and Systems, 13(2): 291-294, 1991.

[9] A. Diwan, K. McKinley, and J. Moss. Type-based alias analysis, In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, pages 106-117, Montreal, Canada, 17-19 June 1998.

[10] C. Dulong. The IA-64 Architecture at Work, IEEE Computer, Vol. 31, No. 7, pages 24-32, July 1998.

[11] C. Dulong, R. Krishnaiyer, D. Kulkarni, D. Lavery, W. Li, J. Ng, and D. Sehr. An overview of the Intel IA-64 compiler. Intel Technology Journal, November 1999.

[12] M. Fernande, and R. Espasa. Speculative alias analysis for executable code, In Proceedings of International Conference on Parallel Architectures and Compilation Techniques, pages 222-231, Charlottesville, Virginia, Sept 2002.

[13] R. Ghiya, D. Lavery, and D. Sehr. 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, pages 47-58, Snowbird, Utah, June 2001.

[14] M. Hind. Pointer analysis: Haven't we solved this problem yet? In ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, pages 54-61, Snowbird, Utah, June 2001.

[15] Y.-S. Hwang, P.-S. Chen, J.-K. Lee, and R. D.-C. Ju, Probabilistic Points-to Analysis, In Proceeding of the Workshop of Languages and Compilers for Parallel Computing, Aug. 2001.

[16] R. D.-C. Ju, J. Collard, and K. Oukbir. Probabilistic Memory Disambiguation and its Application to Data Speculation, Computer Architecture News, Vol. 27, No.1, March 1999.

[17] R. D.-C. Ju, K. Nomura, U. Mahadevan, and L.-C. Wu. A Unified Compiler Framework for Control and Data Speculation, In Proceedings of 2000 International Conf. on Parallel Architectures and Compilation Techniques, pages 157 - 168, Oct. 2000.

[18] R. D.-C. Ju, S. Chan, and C. Wu. Open Research Compiler (ORC) for the Itanium Processor Family. Tutorial presented at Micro 34, 2001.

[19] R. D.-C. Ju, S. Chan, F. Chow, and X. Feng. Open Research Compiler (ORC): Beyond Version 1.0, Tutorial presented at PACT 2002.

[20] R. Kennedy, F. Chow, P. Dahl, S.-M. Liu, R. Lo, and M. Streich. Strength reduction via SSAPRE. In Proceedings of the Seventh International Conference on Compiler Construction, pages 144--158, Lisbon, Portugal, Apr. 1998.

[21] R.Kennedy, S. Chan, S. Liu, R. Lo, P. Tu, and F. Chow. Partial Redundancy Elimination in SSA Form. ACM Trans. on Programming Languages and systems, v.21 n.3, pages 627-676, May 1999.

[22] K. Knobe and V. Sarkar. Array SSA form and its use in parallelization. In Proceedings of ACM Symposium on Principles of Programming Languages, pages 107--120, San Diego, California, January 1998.

[23] J. Knoop, O. Ruthing, and B. Steffen. Lazy code motion. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, pages 224-234, San Francisco, California, June 1992.

[24] J. Lin, T.Chen, W.C. Hsu, P.C. Yew, Speculative Register Promotion Using Advanced Load Address Table (ALAT), In Proceedings of First Annual IEEE/ACM International Symposium on Code Generation and Optimization, pages 125-134, San Francisco, California, March 2003

[25] R. Lo, F. Chow, R. Kennedy, S. Liu, P. Tu, Register Promotion by Sparse Partial Redundancy Elimination of Loads and Stores, . In Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation, pages 26--37, Montreal, 1998

[26] E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. Communications of the ACM, 22(2): 96--103, 1979.

[27] pfmon: ftp://ftp.hpl.hp.com/pub/linux-ia64/pfmon-1.1-0.ia64.rpm

[28] B. Steensgaard. Points-to analysis in almost linear time. In Proceedings of ACM Symposium on Principles of Programming Languages, pages 32--41, Jan. 1996.

[29] R.P. Wilson and M.S. Lam. Efficient context-sensitive pointer analysis for C program. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, pages1-12, La Jolla, California, Jun 18-21, 1995.

[30] Y. Wu and Y. Lee. Accurate Invalidation Profiling for Effective Data Speculation on EPIC processors, In 13th International Conference on Parallel and Distributed Computing Systems, Las Vegas, Nevada, Aug 2000.

Appendix A

Algorithm for the enhanced ϕ insertion which allows data speculation

```
procedure ϕ-Insertion(E)
DF_phis← {};
for each occurrence v of E in program do {
        DF_phis← DF_phis ∪ DF+(v)
        var_phi_list ← {}
        while (v is defined by χ without speculation flags)
            v ←the operand of χ
        if (v is defined by ϕ){
            set_def_phi_recursive(phi(v),var_phi_list);
            DF_phis← DF_phis ∪ var_phi_list;
        }
        Insert ϕ for E according to DF_phis
}
end ϕ-Insertion
procedure set_def_phi_recursive(par_phi, var_phi_list)
if (par_phi ∉ var_phi_list){
  var_phi_list ← var_phi_list ∪ {par_phi}
  for each operand v in par_phi do{
      while (v is defined by χ without speculation flags)
          v ← the operand of χ
      if (v is defined by ϕ)
          set_def_phi_recursive(phi(v), var_phi_list);
   }
}
end set_def_phi_recursive
```

Appendix B

Algorithm for the enhanced CodeMotion step which handles data speculation

```
procedure CodeMotion(E) {
...
   for each occurrence p of expression E in post-order DT
       traversal order do{
        if ( speculative(p) is true  &&
            (p is marked with reload or p is a ϕ operand of a
             ϕ occurrence marked with will_be_avialbe))
             Set_speculative_check_flag ( p )
   }
}
end CodeMotion

procedure Set_speculative_check_flag (p)
q ← avail_def (p)
D ← defining statement of p
if  (the check statement for p not yet generated for D) {
      generate an assignment statement stmt  which is a save
      of the computation E after D;
      speculative_check(stmt) ← advance load check flag for
ld.c
      if (E is an indirect reference){
          a ← the address expression of p
          if (a is defined by a speculative check statement s)
                speculative_check(s) ←advance load check flag
for chk.a
      }
      if (q is real occurrence or inserted occurrence )
          speculative_load(q) ← advance load flag
}
else if (q is ϕ occurrence){
       phi_list ← {}
       Set_speculative_load_flag(phi(q), phi_list)
}
end Set_speculative_check_flag

procedure Set_speculative_load_flag (par_phi,phi_list)
phi_list ← phi_list ∪ {par_phi}
for (each operand q of par_phi){
     if avail_def(q) is a ϕ occurrence and q ∉ phi_list
          Set_speculative_load_flag(q, phi_list)
      else if avail_def(q) is a real or inserted occurrence{
          r ← avail_def (q)
           speculative_load(r) ← advance load flag
       }
}
end Set_speculative_load_flag
```