Proving Safety of Speculative Load Instructions at Compile-Time

David Bernstein
Michael Rodeh
Mooly Sagiv
IBM Israel Scientific Center
the Technion City, Haifa 32000, Israel
Email: bernstn@haifasc3.vnet.ibm.com

Abstract

Speculative execution of instructions is one of the primary means for enhancing program performance of superscalar and VLIW machines. One of the pitfalls of such compile-time speculative scheduling of instructions is that it may cause run-time exceptions that did not exist in the original version of the program.

As opposed to run-time hardware or software interception of such exceptions, we suggest that the compiler will analyze and prove the *safety* of those instructions that are candidates for speculative execution, rejecting the ones that have even a slight chance of causing an exception.

Load (moving a memory operand to a register) instructions are important candidates for speculative execution, since they precondition any follow-on computation on load-store architectures. To enable speculative loads, an algorithmic scheme for proving the safety of such instructions is presented and analyzed. Given a (novel) memory layout scheme which is specially tailored to support safe memory accesses, it has been observed that a significant part of load instructions can be proven safe and thus can be made eligible for speculative execution.

1 Introduction

The recent advent of superscalar and VLIW machines increases the need for aggressive instruction scheduling by optimizing compilers. If previously (for pipelined machines) it was sufficient to reorder instructions at the basic block level ([HG83], [War90]), it is evident now that for the newer machines, instructions have to be moved well beyond basic block boundaries. A few such efforts were described in [Ell85, EN89, GS90, BR91].

It turns out that very often, to further improve the utilization of machine resources, instructions have to be scheduled *speculatively*, i.e., moved ahead of a preceding branch to a place were it is not yet determined in the program that such instructions have to be executed at all. Previously, speculative execution was considered in the context of moving loop-invariants out of loops [ASU85], while recently, due to the evolution toward superscalar and VLIW machine designs, this type of transformations is being exploited in a much broader scope of code motions. In case the compiler guesses right the direction of the branch over which a speculative instruction is moved, such instruction computes useful results; otherwise the compiler must make sure that these results will not be used in the subsequent execution of the program.

One of the main problems of speculative execution of instructions, which is the focus of this paper, is that they may cause *program exceptions* which were not supposed to happen if these instructions were executed in their original (non-speculative) places. Kennedy first raises the problem of proving *safety* of such instructions [Ken72], with the motivating example of moving a division loop invariant instruction out of a loop (which is a special case of speculative code motion). The reasons for program exceptions are diverse: arithmetic operations may result in an overflow, memory access instructions (loads and stores) may reference an invalid memory address, etc. In most of the previous work it was suggested that prevention of such exceptions must be supported by hardware or software in run-time, some of them even proposed to disable the exceptions of speculative instructions all together (for more details, see Section 8).

Our proposal is different than all previous work on speculative scheduling and it extends Kennedy's technique of [Ken72]. We suggest to determine, at compile-time, which instructions are *safe*, i.e., which instructions will never cause exceptions that would not have occurred in the original version of P. Then, only safe instructions will be candidates for speculative execution. Using this approach, in the optimized version of the program we have no problem to allow the exceptions to occur, in a way similar to the original (non-optimized) program. Thus, the advantage of our approach is that it both preserves the *debuggability* of the program (meaning - exceptions do not have to be masked) and does not require any hardware support for speculative execution. In this paper we concentrate on proving safety of load instructions (which is one of the most important classes of speculative instructions), even though our techniques can be applied to additional types of instructions as well. In particular, this paper does not address the issue of profitability of speculative scheduling which may be affected by the underlying architecture (e.g., an increased number of cache misses, page faults, and register pressure), neither it deals with guaranteeing correct results for speculative code motions. These problems were partially covered in [Ell85, BR91].

The crucial question is: What fraction of speculative load instructions can be proven safe at compile time? Our experience is that, for a set of benchmarks we considered, provided a certain model of memory layout is assumed, a significant part of load instructions can be proven safe at compile-time. These results are summarized in Section 7.

In our set-up, the address of a load instruction is defined by (the contents of) a machine register plus a displacement. Our first result is a linear-time algorithm for determining the safety of a load instruction whose address is defined by register r, using the information about the existence of different memory-access instructions (loads or stores) in different points of the program whose addresses are defined by the same register r. This algorithm does not take into account the contents of the registers, i.e., the assumption is that the value of r is unchanged during the portion of the program under inspection. In his original paper [Ken72], Kennedy also describes an iterative backward algorithm for proving safety. His algorithm is less accurate than ours, since we also use forward information and safety of certain register assignment statements (for a technical comparison see Section 8).

The second result is an efficient algorithm for proving safety in case the value of r may change during the execution of the program by statements like $r = r + c$, where c is a constant. Here we take advantage of the observation that, if accessing location (r) is valid ((r) is the contents of r), then the access to location $(r) + k$, where k is the size of the page in the memory system, is valid as well. In Section 5, we describe the requirements on the memory layout that are needed to make this assumption legitimate.

The accuracy of our algorithms depends on two assumptions. The first assumption is that every execution path of the program can be taken. The second assumption is that there are no program statements, like $r_1 = r_2 + c$, or $r_1 = r_2 + r_3$. In fact, we show that the problem of proving safety in the simple case when only register transfer statements (i.e., statements of the form $r_1 = r_2$) are allowed is CO-NP-hard. For this case we do provide an iterative conservative approximation algorithm (see [Kil73, KU76]).

The paper is organized as follows. In the next section we start with the definition of the problem. Then, in Section 3, the first algorithm is presented. Register transfer statements are discussed in Section 4 (including CO-NP-hardness result). In Section 5 we define our novel memory layout scheme. Then, in Section 6, the second algorithmic result is presented. Some experimental results are described in Section 7. Section 8 contains a discussion on related work.

2 Definition of the Problem

Here we formally state the problem of proving safety, so as to lay a basis for discussion. In the rest of this paper, P is a terminating low level program that uses a set of registers R. We assume that references to memory locations are done by load instructions of the form *load* $r', (r)$. By convention, we say that such instruction includes a memory reference to a location whose address is the contents of r (denoted by *ref*(r)).

Let i be an instruction that includes *ref*(r) and *pt* a point in P. We say that the insertion of i before *pt* is *safe* if it does not cause an exception that would not have occurred in the original version of P. The speculative code motion that was presented in the introduction can be viewed as consisting of two separate actions, namely, an instruction is deleted from one point in P and inserted into another point. In the rest of the paper we will deal only with proving the safety of insertions of instructions into new places of the program (as it is reflected in the above definition of safety). Notice that deletions of instructions from P cannot create new exceptions, so it is not related to the problem of safety. The discussion of the data dependency correctness of deletions and insertions is out of the scope of this paper. However, the combination of insertion and deletion, while taking into account data dependency, enables instruction motions, an essential element of instruction scheduling.

The above definition of safety is implicit since the notion of exception was not defined. Let π be an execution sequence of P and let *pt* be a point in P along π. Let h be the value of r at at point *pt* of π. Then, the insertion of *ref*(r) before *pt* is *safe* along π if one of the following is true:

1. there is an instruction in π which refers to h as address.

2. h is a valid address.

This definition implies that if h is referred as address, the insertion of *ref*(r) cannot create new exceptions; otherwise, it is safe only when h is known to be a valid address. The notion of a valid address is defined by the programming language and/or the operating system[1].

We say that the insertion of *ref*(r) before *pt* is *strongly safe* if it is safe at *pt* along every execution sequence of P in which *pt* appears. In Sections 3 and 4, we consider the algorithmic problem of

[1]In languages like C, the storage allocation of heap objects is only defined as part of the operating system support.

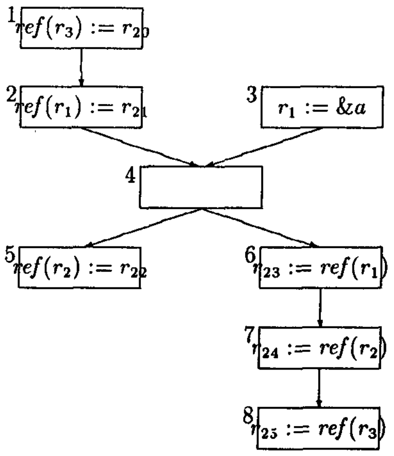

Figure 1: A program control flow graph for a conditional construct

finding strongly safe references. In Section 5, we extend the class of valid addresses by providing a new memory layout scheme, and refine the algorithms accordingly.

3 A Path Analysis Algorithm

In the rest of this paper, $G = (V, E)$ is the program control flow graph of the given program fragment P; each vertex $v \in V$ represents a single instruction in the program, and there is an edge $(v, u) \in E$ if the program control can flow directly from vertex v to vertex u (e.g., see [WZ85]). Thus, every point in P corresponds to a vertex in G. A vertex v can use the contents of register $r \in R$, and can assign a new value to r, as the last operation in v, denoted by $\mathit{def}(r)$. For example, Figure 1 contains the program control flow graph for a program fragment which corresponds to a conditional construct. Instruction 7 contains $\mathit{def}(r_{24})$.

A path π in G which starts at a vertex $v \in V$ is *r-definition free* for $r \in R$ if every vertex u on π other than v does not include $\mathit{def}(r)$. Such a path is *maximal* if every successor of the last vertex in π contains $\mathit{def}(r)$. In the program control flow graph of Figure 1, the paths $(1,2,4,5)$ and $(3,4,6,7,8)$ are r_1 maximal definition free and the path $(1,2,4,6,7)$ is not. Since the only definition of r_1 appears in a source vertex 3, all the paths in G are r_1-definition free. Figure 2 contains another program control flow graph for a loop construct. This loop is a simplified version of a traversal on a linked list. Assuming that the next pointer is located in the first field of the list records, instruction 4 increments r_1 to the next list element. Instruction 6 contains $r_3 := ref(r_2)$ but should have contained $r_3 := ref(r_2 + k)$ where k the relative place of the data within the record. The path $(2,4,5)$ in this program control flow graph is not r_1-definition free.

In the rest of this section, we present a simplified algorithm for proving safety by only considering definition free paths. An r-definition free path π starting at v is *$\mathit{ref}(r)$ safe* if one of the following conditions holds:

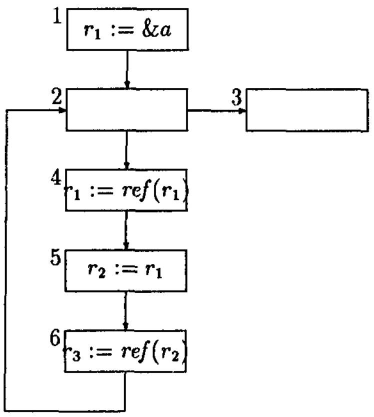

Figure 2: A control flow graph of a loop construct

1. v does not include *def*(r) and one of the vertices in π includes *ref*(r).

2. v contains an instruction which assigns a valid address to r or a vertex in π other than v includes *ref*(r).

If a path from v is *ref*(r) safe then the insertion of *ref*(r) is safe before every vertex in π other than v along every execution path of P which contains π. The path $(1,2,4,6,7,8)$ in the program control flow graph of Figure 1 is *ref*(r_3) safe but $(3,4,5)$ is not.

Lemma 3.1 *Let $v \in V$ and $r \in R$. Then, the insertion of ref(r) before v is strongly safe if the following conditions are met:*

1. *For every source vertex v_0 of G and for every maximal r-definition free path π from v_0 which goes through v, π is ref(r) safe.*

2. *If $u \in V$ contains a definition of r then every maximal r-definition free path π from u which goes through v, π is ref(r) safe.*

$\square$

The r_3-definition free path $(3,4,5)$ in the program control flow graph of Figure 1 is not *ref*(r_3) safe. This indicates that the insertion of *ref*(r_3) is not strongly safe before any vertex in this path. In particular, it is not strongly safe before vertex 4 which is a candidate place for speculative execution of the instruction $r_{25} := ref(r_3)$.

We now construct an algorithm which checks the conditions of Lemma 3.1 for $r \in R$ and every vertex $v \in V$. Thus, the algorithm can be applied separately for every $r \in R$.

First, let us define $G_1[r]$ to be the graph which includes the definition free paths for r in G. Technically, let s and t be new vertices. Let $V_d[r] \subseteq V$ be the vertices which include *def*(r). Since a vertex $v \in V_d$ can contain both *ref*(r) and *def*(r), v is duplicated in $G_1[r]$; this is done by adding

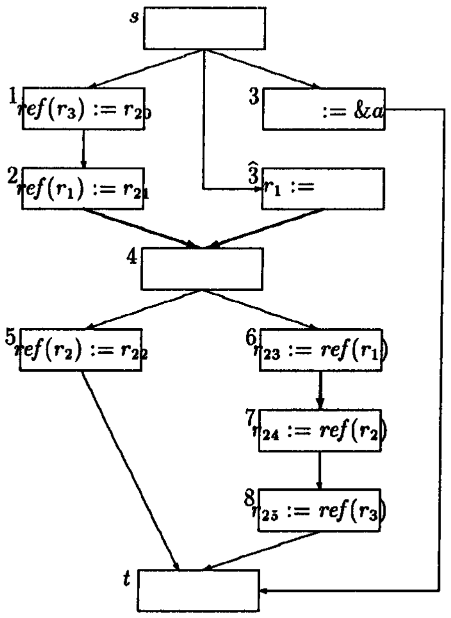

Figure 3: The graph $G_1[r_1]$ for the example of Figure 1

a second copy of v denoted $\hat{v}$. Thus, $G_1[r] = (V_1[r], E_1[r])$, where $V_1[r] = V \cup \{\hat{v} | v \in V_d[r]\} \cup \{s, t\}$ and $(u, v) \in E_1[r]$ if one of the following conditions holds:

1. $u = s$ and v is a source in G.
2. $u = s$ and $v = \hat{w}$ where $w \in V_d[r]$.
3. $(u, v) \in E$ where $u \notin V_d[r]$.
4. $u = \hat{w}$, $w \in V_d[r]$ and $(w, v) \in E$.
5. $u \in V_d[r]$ and $v = t$.
6. u is a target in G and $v = t$.

Thus, any path from s to t in $G_1[r]$ corresponds to a path which may start with a definition of r and is subsequently r-definition free. Figure 3 contains $G_1[r_1]$ for the program control flow graph of Figure 1. Since only r_1 is assigned in this program, $G_1[r_2] = G_1[r_3] = G_1[r_4]$ is the graph which consists of connecting s to 1 and 3, and connecting 5 and 8 to t. Figure 4 contains $G_1[r_1]$ for the program control flow graph of Figure 2.

Now, let $G_2[r]$ be the graph obtained from $G_1[r]$ by deleting edges which emanate from vertices with an evidence of safety, i.e., $G_2[r] = (V_1[r], E_2[r])$ where $E_2[r] \subseteq E_1[r]$ and $(u, v) \in E_1[r] - E_2[r]$ if and only if $u \in V$ and u contains an instruction which assigns a valid address to r or $v \in V$ and v contains *ref*(r). The graph $G_2[r]$ may be used to check the safety of *ref*(r) based on the following lemma.

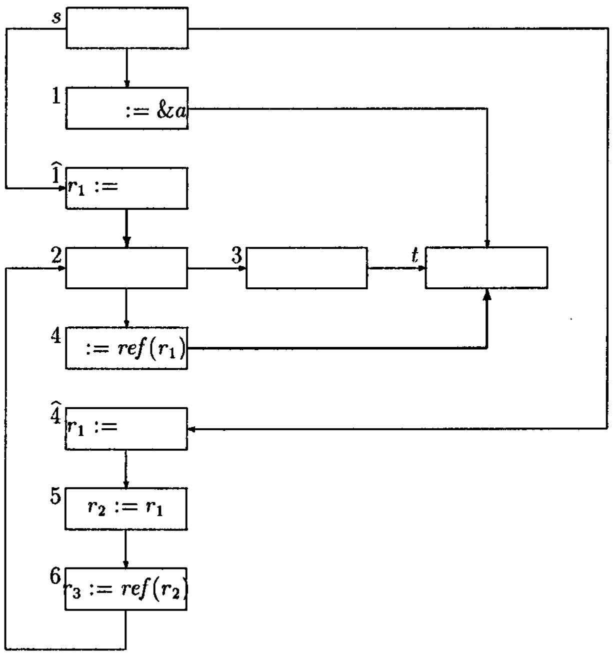

Figure 4: The graph $G_1[r_1]$ for the example of Figure 2

Lemma 3.2 *For a vertex $v \in V$, there does not exist a path in $G_2[r]$ from s to t which goes through v if and only if the conditions of Lemma 3.1 for v and r are met.* □

Thick lines in Figure 3 and Figure 4 denote edges in $G_1[r_1]$ which do not appear in $G_2[r_1]$. Figure 3 shows that the insertion of *ref*(r_1) in strongly safe before every vertex but 3, since the only path from s to t in $G_2[r_1]$ is $(s, 3, t)$. On the other hand, in Figure 4, the insertion of *ref*(r_1) is not detected as strongly safe before any vertex but 4, since there exist paths from s to t either by $(s, 1, t)$, or by $(s, \hat{4}, 5, 6, 2, 3, t)$.

Lemma 3.2 suggests that an ordered depth first search (DFS) algorithm may be applied to $G_2[r]$ starting at s so as to detect safety. Initially, all the vertices are marked as allowed to have insertions of *ref*(r) which are (strongly) safe. When the algorithm backtracks from t, it marks its predecessors vertices as being unsafe for the insertion of *ref*(r).

The correctness of the path analysis algorithm stems from Lemmas 3.1 and 3.2.

The complexity of the path analysis algorithm for $r \in R$ is $O(|E_2[r]|) = O(|V| + |E|)$ and thus linear in the size of the program.

The conditions in Lemma 3.1 are sufficient, but not necessary to prove that the insertion of *ref*(r) is strongly safe. Therefore, the path analysis algorithm yields conservative, but not necessarily accurate results. For example, as mentioned above, the insertion of *ref*(r_1) is not detected as strongly safe before vertex 2 in Figure 2 although it is. This insertion is safe due to the instruction in vertex 6 and the fact that $r_1 = r_2$ there. Indeed, the conditions of Lemma 3.1 are syntactical in the sense that values are not taken into account. Only for a class of programs in which for every instruction i that includes $def(r)$, the validity of the address that is assigned to r is known from the properties of i (i.e., there is no case in which, looking on i, we are not sure if the assigned address is valid), the path analysis algorithm is exact, i.e., the conditions of Lemma 3.1 are necessary.

4 Tracking Values

4.1 The Negative Result

Theorem 4.1 *The problem of detecting that a reference to a register is safe is CO-NP hard even under the following assumptions on the analyzed program P:*

1. *all the control flow paths in P are executable;*
2. *the control flow graph does not contain loops;*
3. *the only instructions in P are load instructions and register transfer instructions of the form $r_1 := r_2$.*

Proof: By reduction from the satisfiability problem. For example, Figure 5 contains a program for the satisfiability of an example formula. □

The reader may notice the similarity between Theorem 4.1 and negative results of solving data flow problems in presence of aliasing (e.g., [Mye81, Lar89, SFRW90]). In fact, the essence of all these results is that to track alias effects accurately one needs to keep track of all the possible sets of address variables which are equal.

```
if ⋯
        then $x_1 := f$
        else $x'_1 := f$
if ⋯
        then $x_2 := f$
        else $x'_2 := f$
if ⋯
        then $x_3 := f$
        else $x'_3 := f$
if ⋯ { code for $x_1 \vee x_2$ }
        then $r := ref(x_1)$
        else $r := ref(x_2)$
if ⋯ { code for $x'_1 \vee x'_2 \vee x_3$ }
        then $r := ref(x'_1)$
        else if ⋯
                    then $r := ref(x'_2)$
                    else $r := ref(x_3)$
```

Figure 5: $ref(f)$ is safe at the program entry if and only if $(x_1 \vee x_2) \wedge (x'_1 \vee x'_2 \vee x_3)$ is not satisfiable

4.2 Conservative Approximations

We now develop an iterative algorithm which is more accurate than the path analysis algorithm of Section 3. The problem of detecting that the insertion of $ref(r)$ before v is safe along an execution path π can be divided into two problems. In the *forward* problem the path segment from the beginning of π to v is analyzed, while in the *backward* problem the path segment of π from v to the end of π is analyzed. We say that the insertion of $ref(r)$ before $v \in V$ is *forward safe* along an execution sequence π of P if the value of r at v on π is either referred to as an address or is known to be a valid address. Such insertion before v is *backward safe* along π if the value of r at v on π is referred as address. Notice that in this case the value of r may be assigned prior to v on π. Thus, to determine that the insertion is backward safe along π, the path segment of π from the beginning of the program to v also need to be considered.

The insertion of $ref(r)$ before $v \in V$ is *forward* (respectively *backward*) *strongly safe* if it is forward (respectively backward) safe along every execution path which goes through v. We have the following simple lemma.

Lemma 4.2 *An insertion of $ref(r)$ before $v \in V$ is strongly safe if and only if it is either forward strongly safe or it is backward strongly safe.* □

In the rest of this section we present a forward iterative algorithm to detect that an insertion is forward strongly safe. A backward iterative algorithm for detecting that an insertion is backward safe can be specified, in a similar fashion. The combination of both algorithms provides an approximation

vertex	val_b						val_a					
	r_1	r_2	r_3	$ref(r_1)$	$ref(r_2)$	$\&a$	r_1	r_2	r_3	$ref(r_1)$	$ref(r_2)$	$\&a$
1	1	2	3	4	5	6	1	2	3	4	5	1
2	1	2	3	4	5	6	1	2	3	4	5	6
3	1	2	3	4	5	6	1	2	3	4	5	6
4	1	2	3	4	5	6	1	2	3	4	5	6
5	1	2	3	4	5	6	1	1	3	4	5	6
6	1	1	3	4	5	6	1	1	3	4	3	6

Table 1: Value numbers for the example of Figure 2

scheme for proving safety which is more accurate than the path analysis algorithm presented in Section 3.

Two expressions in the program e_1 and e_2 are *equivalent* before $v \in V$ if their values are equal before v at every execution path of P which goes through v. If two expressions e_1 and e_2 are equivalent, then any evidence for the safety of e_1 is also an evidence for the safety of e_2. Thus, an algorithm which finds a conservative approximation of equalities may be useful in proving safety. In the sequel, we shall use the information on equivalences which is represented by global value numbers, i.e. an association of hash values with symbolic expressions. Thus, e_1 and e_2 are equivalent before v if $val_b[v](e_1) = val_b[v](e_2)$ where $val_b[v](e)$ is the value number of e before v. The notion of equivalence after v (denoted by $val_a[v](e)$) is similarly defined. Efficient algorithms for computing global values numbers are known (e.g., [RL77, AWZ88, RWZ88]). Table 1 contains possible value numbers for the example of Figure 2.

Given value numbers of every $v \in V$, we now sketch an iterative algorithm for computing forward strong safety. The algorithm maintains at every $v \in V$ and for every value val before v a boolean value $s_b[v](val)$ which describes the strong safety of this number before v. For convenience, the algorithm also maintains as auxiliary information $s_a[v](val)$ which describes the strong safety of this number after v. Initially, for every source vertex v of G, $s_a[v](val) = true$ and $s_b[v](val)) = true$ if and only if there exists an expression e such that e always holds a valid address and $val = val_b[v](e)$. For any other $v \in V$, the initialization is $s_b[v](val) = s_a[b](val) = true$. The algorithm iterates on G using a DFS order and stops when no new information is derived. Let v be an assignment statement of the form $r := e$. Let $r_1, r_2, \ldots, r_n$ be the registers referred as addresses in e. Then, $s_b[v](val)$ is computed in the iteration using the equation:

$$s_b[v](val) = \begin{cases} true & \exists i : 1 \leq i \leq n, val = val_b[v](r_i) \\ \wedge\{s_a[u](val_a[u](e)) \mid (u,v) \in E, \exists e, val_b[v](e) = val\} & \text{otherwise} \end{cases} \tag{1}$$

Also, $s_a[v](val)$ is computed by:

$$s_a[v](val) = \begin{cases} s_b[v](val_b[v](e)) & val = val_a[v](r) \\ s_b[v](val) & \text{otherwise} \end{cases} \tag{2}$$

The treatment of other type of instructions is similar. Table 2 exemplifies the application of the iterative algorithm to the program of Figure 2, with the value numbers of Table 1. Undefined value numbers are denoted by ϕ. We see that two iterations are sufficient in this case and that the insertion of $ref(r_1)$ is detected as strongly safe before vertex 2 of Figure 2.

		s_b						s_a					
iteration	**vertex**	1	2	3	4	5	6	1	2	3	4	5	6
0	1	f	f	f	f	f	t	t	t	t	t	t	t
0	2, 3, 4, 5, 6	t	t	t	t	t	t	t	t	t	t	t	t
1	1	f	f	f	f	f	t	t	f	f	f	f	ϕ
1	2	t	f	f	f	f	t	t	f	f	f	f	t
1	3	t	f	f	f	f	t	t	f	f	f	f	t
1	4	t	f	f	f	f	t	f	f	f	f	f	t
1	5	f	f	f	f	f	t	f	ϕ	f	f	f	t
1	6	t	ϕ	f	f	f	t	f	ϕ	f	f	ϕ	t

Table 2: The safety vectors for the example of Figure 2

```
if ···
    then $r_1 := r_2$
    else $r_1 := r_3$
*load* $r', (r_2)$
*load* $r', (r_3)$
```

Figure 6: A program which demonstrates the inaccuracy of the conservative algorithm

Lemma 4.3 *When the forward iterative algorithm terminates, for every $r \in R$ and $v \in V$ s.t. $s_b[v](val_b[v](r))$ holds, the insertion of ref(r) before v is forward strongly safe.* □

The opposite direction of Lemma 4.3 does not hold. The first reason is that value numbers are not always exact, i.e., it is possible that two expressions are equivalent and yet they get two different value numbers. Moreover, even when the value numbers are accurate, the algorithm may fail to detect safety. For example, in the program fragment of Figure 6, the insertion of $ref(r_1)$ as the last statement in the program is forward strongly safe and yet will not be detected as such by our algorithm. The reason is that the algorithm does not use the fact that r_1 is either r_2 or r_3, and thus since both of these registers are referred, $ref(r_1)$ becomes safe.

5 Memory Organization

In this section, the domain of valid addresses is extended by suggesting a new memory layout organization. In Section 6, by taking advantage of this extended memory layout support, we improve the algorithms of Sections 3 and 4, so as to allow more instructions to be scheduled speculatively without causing memory exceptions.

The first simple but very important type of memory layout support is to assume that address 0 is allowed for access by load instructions. This appears to be extremely useful, since the usual interpretation of **nil** pointers is 0. Thus, a memory access through a **nil** pointer references memory at address 0. For example, consider the following piece of code:

$$if\ (c \neq 0)\ then\ a = *c; \qquad (3)$$

The insertion of $a = *c$ before the comparison $c \neq 0$ is strongly safe under the assumption that zero is a valid address. This assumption may have negative impact on the debuggability of programs, since memory accesses through **nil** pointer will be allowed. The main motivation for this extension to our approach is that all the Unix implementations that we have checked allow this kind of read access to 0 address.

It is evident that in practice, many of the memory references are within small distances of other memory references. To take advantage of this property in a general context, we suggest to use page padding, i.e., to allocate dummy pages on both sides of the data segment(s), as well as on both sides of address 0, and allow these pages to be accessed by load instructions. This type of extended memory layout support has a minor negative impact on debuggability: only in some rare cases of dangling references that happen to access variables that are mapped closed to the end of the data segment, exceptions may be lost.

Intuitively, the above assumption on page padding implies that if we find in a program a memory reference to address a, then the accesses to all the addresses in the range $[a - k, a + k]$, for some fixed k (e.g., k is the size of the page in the memory system), will not cause exceptions. Also, by the same padding property, the memory accesses in the range of $[-k, k]$ are allowed as well. (Notice that to support memory accesses to negative virtual addresses, a special type of operating system support is required.)

To put this extended memory layout support in a formal way, we modify the definition of safety on an execution path from Section 2. Let π be an execution sequence of P, and let pt be a point in P along π. Let h be the value of r at π in pt. Then, the insertion of *ref*(r) before pt is *safe* along π if there exists l such that $|l - h| \leq k$, and one of the following conditions is true: .

1. there is an instruction in π which refers to l as address.

2. l is a valid address.

6 Exploiting the Improved Memory Layout

The notion of valid addresses has served as a parameter of both the path analysis algorithm presented in Section 3 and the conservative algorithms of Section 4.2. Thus, these algorithms can easily handle the refinements to valid addresses as those that were presented in Section 5.

Limited form of condition statement can also be handled. For example, the conditional statement in (3) (see Section 5) may be handled by inserting a dummy *ref*(c) in the empty else clause of this statement. This dummy assignment uses the fact that is legal to refer to c when $c = 0$. After this assignment has been inserted, the path analysis algorithm will find that the insertion of *ref*(c) is strongly safe before the whole conditional statement.

Supporting page padding is somewhat more complicated than that. A simple conservative approximation is obtained by allowing displacements as part of the specification of an address for a memory reference. Here, instructions of the form: $ref(r + \Delta)$, where $-k \leq \Delta \leq k$, are allowed. The path analysis algorithm will interpret such instruction as a reference to address r, and will use this information to prove safety. By the extended definition of safety in Section 5, Lemma 3.1 remains true.

The main problem with this approach is that the assignment instructions of the form $r_1 := r_2 + \Delta$ are always interpreted as unsafe. For example, if r_2 is referred prior to such an instruction v, then, for every l such that $-k - \Delta \le l \le k - \Delta$, the insertion of $ref(r_1 + l)$ is strongly safe after v. In the rest of this section, the algorithm of Section 4.2 is refined to handle such instructions.

Since Lemma 4.2 remains true for the extended definition of safety, we now refine the conservative forward and backward algorithms of Section 4, so as to handle instructions of the form $r_1 := r_2 + \Delta$ where Δ is an integer literal constant.

Recall that the data structure in the algorithm of Section 4.2 is the safety boolean vectors $s_b[v](val)$ and $s_a[v](val)$. To handle page padding, these boolean vectors are replaced by sets of integers denoted by capital letters $S_b[v](val)$ where $z \in S_b[v](val_b[v](e))$ if the insertion of $ref(r + z)$ before v is strongly safe. The sets $S_a[v](val)$ after v are similarly defined.

For a set of integers S and an integer z, we define $S + z$ as follows:

$$S + z \stackrel{\mathrm{def}}{=} \{z' + z | z' \in S\}.$$

Also, let $S_k = \{z : -k \le z \le k\}$ and $\top$ be an element which denotes the universe set of integers.

For every source vertex v of G, the new initialization is $S_a[v](val) = \top$ and $S_b[v](val)) = S_k$ if there exists an expression e such that e always holds valid addresses and $val = val_b[v](e)$, and otherwise $S_b[v](val) = \phi$. For any other $v \in V$, the initialization is $S_b[v](val) = S_a[b](val) = \top$.

Let v be an assignment statement of the form $r := e_0 + z$ where z is an integer literal. Let $r_1, r_2, \ldots, r_n$ be the registers referred as addresses in e. Then, $S_b[v](val)$ is computed in the iteration using the equation:

$$S_b[v](val) = \begin{cases} \cap\{S_a[u](val_a[u](e)) | (u,v) \in E, \exists e, val_b[v](e) = val\} \cup S_k & \exists i : 1 \le i \le n, val = val_b[v](r_i) \\ \cap\{S_a[u](val_a[u](e)) | (u,v) \in E, \exists e, val_b[v](e) = val\} & \text{otherwise} \end{cases}$$

Also, $S_a[v](val)$ is computed by:

$$S_a[v](val) = \begin{cases} S_b[v](val_b[v](e_0) - z & val = val_a[v](r) \\ S_b[v](val) & \text{otherwise} \end{cases}$$

The treatment of other types of instructions is similar.

Lemma 6.1 *When the forward iterative algorithms terminates, for every $r \in R$, $v \in V$ and $z \in S_b[v](val_b[v](r))$, the insertion of $ref(r + z)$ before v is strongly safe.* $\square$

Similarly to the forward algorithm, a backward iterative algorithm can be suggested.

7 Experimental results

Here we present experimental results for proving safety of speculative loads in the context of a prototype for *global scheduling* for the IBM RS/6000 machine [BR91]. The implemented algorithm yields results which are less accurate than the path analysis algorithm of Section 3, but they are comparable. Also, for efficiency, the analysis in the implemented algorithm was done locally. For example, for proving safety of a load instruction in a loop, only program statements in this loop were considered. The implemented algorithm assumes the memory layout organization that was described in Section 5.

We have evaluated this algorithm on a set of benchmarks (written in C) as follows:

program	total	safe	p-unsafe	p-safe
LI	1286	81%	1%	18%
EQNTOTT	640	43%	6%	51%
ESPRESSO	2759	41%	2%	57%
GCC	6865	55%	2%	43%

Table 3: Experimental results for safety of speculative loads

1. LI: LISP interpreter,
2. EQNTOTT: translation of Boolean equations into truth tables,
3. ESPRESSO: logic minimization,
4. GCC: the GNU C compiler,

The results are presented in Table 3 which provides the following information:

1. **total**: the total number of load instructions considered for proving their safety. Notice that not all of these instructions will be subsequently scheduled for speculative execution.
2. **safe**: a fraction of load instructions that were proved safe.
3. **p-unsafe** (probably unsafe): a fraction of load instructions which refer to addresses loaded from unknown memory places. It is in general hard, if not impossible, to prove safety of these instructions.
4. **p-safe** (potentially safe): a fraction of load instructions that were not proved safe, but are not **p-unsafe**. These instructions are natural candidates for improving the accuracy of the algorithms for proving safety.

It is worthwhile to notice that there is a considerable fraction of load instructions that were not proved safe, but usually they do not cause exceptions when scheduled speculatively (**p-safe** column). This means that, by extending the current algorithm in a way similar to the suggestions in Section 4.2 and using more global information, we should be able to prove safety of even a larger fraction of speculative loads than was shown in Table 3.

8 Related work

Kennedy's safety algorithm in [Ken72] determines that the insertion of a general expression e is safe before a vertex v, i.e., the computation of e before v will not raise new exceptions. This is done by scanning the program iteratively in a backward direction. An expression e is detected as safe before a vertex v only if all the paths after v contain a computation of e. Since Kennedy's algorithm does not take into account values, it is comparable to the path analysis algorithm of Section 3. In the sequel, we explain why our path analysis algorithm is more accurate than Kennedy's.

The path analysis algorithm can be also extended to find the safety of a general expression e. The graph $G_1[e]$ for an expression e is defined by connecting every definition of an argument in e into the vertex t. The edges in $G_1[e] - G_2[e]$ are ones which either contain the usage of e, or a safe definition (like an assignment of a valid address). Of course, one needs to reformulate the set of safe definitions, according to the particular expression and exception. For example, if one wishes to determine that x/y does not raise a divide by zero exception, then any assignment to y of the form $y := c$ where $c \neq 0$ may be considered as safe. The path analysis algorithm identifies more safe expressions (and in particular expressions of the form *ref*(r)) since:

1. The path analysis algorithm also takes into account forward information. For example, if *ref*(r) appears in all the paths to a certain vertex, then the path analysis algorithm will determine that *ref*(r) is strongly safe, whereas Kennedy's algorithm will not.

2. The path analysis algorithm handles safe definitions which are considered unsafe by Kennedy. Kennedy's algorithm may be also modified to handle safe definitions.

Proposing a different direction for handling exceptions, Hennessy [Hen81] suggests to annotate the program, telling where exceptions may happen, so as to disable optimization on those parts of the program.

Considering the safety problem of speculative execution, in [CMC$^+$91] it was suggested to have special non-interruptible machine opcodes for speculative instructions which will smooth all the exceptions resultant by them. The disadvantage of this approach is that we lose the debuggability of the program. For example, it may happen that in its original place an instruction was causing an exception (say, because of a program bug); then after it was moved speculatively, no exception is raised.

Alternatively, the interrupt handling routine of the operating system can be modified, so as to intercept the exceptions at run-time. The compiler is supposed to record the addresses of speculative instructions in a place that is accessible from the interrupt routine. Then, this routine can determine if the exception was caused by a speculative instruction and handle it respectively. The advantage of this approach is that it does not require special opcodes for speculative instructions, but it suffers of the same problem of missed exceptions.

Yet another approach is advocated in [Ebc88]. There, in addition to having non-interruptible opcodes for speculative instructions, there is a special bit for every machine register which is set when the contents of the register is invalid. When a speculative instruction causes an exception, it is not raised, but the result register of the instruction is marked as invalid. This architecture allows to proceed with computing arithmetic instructions whose operands are invalid (if one of the operands is invalid, the result is also invalid). Only when an instruction that has side-effects (like store to memory, branch, etc.) uses an invalid operand, an exception is raised. This approach improves on the previous two, since it does not miss program exceptions. However, it does require significant hardware support and run-time handling of speculative instructions.

Finally, in [SLH90] a massive hardware support for speculative instructions was suggested. It was proposed there that the hardware will not commit on the results of the speculative instructions and will not raise exceptions resultant by them until the direction of the branch over which these instructions were moved is known. This approach allows to determine the exact reason and place of the exceptions caused by speculative execution. Being powerful by its nature, this approach is expensive in both the real-estate on the chip required to implement the hardware support as well as the run-time required to maintain the results of the speculative instructions.

9 Conclusions

Safe loads using the extended memory layout approach point to a very interesting research direction, in which the domain of legal memory accesses is extended beyond the domain which is needed for correct execution of the program. By supporting the extended memory layout, new opportunities for more efficient code are created. It is not clear whether other elements of a given computing model may exhibit the same property, e.g., can their domain be extended so as to allow new optimization techniques.

Acknowledgments

We would like to thank Reinhard Wilhelm for pointing out the work of Kennedy. We would also like to thank Marty Hopkins for numerous useful suggestions, and Doron Cohen and Yuval Lavon for the help in the implementation of the algorithms.

References

[ASU85] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1985.

[AWZ88] B. Alpern, M.N. Wegman, and F.K. Zadeck. Detecting equality of variables in programs. In *ACM Symposium on Principles of Programming Languages*, pages 1–11, 1988.

[BR91] D. Bernstein and M. Rodeh. Global instruction scheduling for superscalar machines. In *SIGPLAN Conference on Programming Languages Design and Implementation*, pages 241–255, 1991.

[CMC+91] P.P. Chang, S.A. Mahlke, W.Y. Chen, N.J. Warter, and W.W. Hwu. IMPACT: An architectural framework for multiple-issue processors. In *IEEE Conference on Computer Architecture*, pages 266–275, 1991.

[Ebc88] K. Ebcioglu. Some design ideas for a VLIW architecture for sequential-natured software. In *IFIP Conference on Parallel Processing*, 1988.

[Ell85] J.R. Ellis. *Bulldog: A Compiler for VLIW Architectures.* PhD thesis, Yale University, February 1985.

[EN89] K. Ebcioglu and T. Nakanati. A new compilation technique for parallelizing regions with unpredictable branches on a VLIW architecture. In *Workshop on Languages and Compilers for Parallel Computing*, 1989.

[GS90] R. Gupta and M.L. Soffa. Region scheduling: An approach for detecting and redistributing parallelism. *IEEE Transactions on Software Engineering*, 16(4):421–431, 1990.

[Hen81] J.L. Hennessy. Program optimization and exception handling. In *ACM Symposium on Principles of Programming Languages*, pages 200–206, 1981.

[HG83] J.L. Hennessy and T. Gross. Postpass code optimization of pipeline constraints. *ACM Transactions on Programming Languages and Systems*, 5:422–448, 1983.

[Ken72] K. Kennedy. Safety of code motion. *Intern. J. Computer Math.*, 3:117–130, 1972.

[Ken81] K. Kennedy. A survey of data flow analysis techniques. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 1, pages 5–54. Prentice-Hall, 1981.

[Kil73] G.A. Kildall. A unified approach to global program optimization. In *ACM Symposium on Principles of Programming Languages*, pages 194–206, 1973.

[KU76] J.B. Kam and J.D. Ullman. Global data flow analysis and iterative algorithms. *Journal of the ACM*, 23(1):158–171, 1976.

[Lar89] J.R. Larus. *Restructuring Symbolic Programs for Concurrent Execution on Multiprocessors.* PhD thesis, University of California, 1989.

[Mye81] E.W. Myers. A precise inter-procedural data flow algorithm. In *ACM Symposium on Principles of Programming Languages*, pages 219–230, 1981.

[RL77] J.H. Reïf and H.R. Lewis. Symbolic evaluations and the global value graph. In *ACM Symposium on Principles of Programming Languages*, pages 104–118, 1977.

[RWZ88] B.K. Rosen, M.N. Wegman, and F.K. Zadeck. Global value numbers and redundant computations. In *ACM Symposium on Principles of Programming Languages*, pages 12–27, 1988.

[SFRW90] S. Sagiv, N. Francez, M. Rodeh, and R. Wilhelm. A logic-based approach to data flow analysis problems. In P. Deransart and J. Maluszynkski, editors, *LNCS 456, 2nd Workshop on Programming Language Implementation and Logic Programming.* Springer-Verlag, 1990.

[SLH90] M.D. Smith, M.S. Lam, and M.A. Horowitz. Boosting beyond static scheduling in a superscalar processor. In *IEEE Conference on Computer Architecture*, pages 344–354, 1990.

[War90] H. Warren. Instruction scheduling for the IBM RISC System/6000. *IBM Journal on Research and Development*, pages 85–92, 1990.

[WZ85] M.N. Wegman and F.K. Zadeck. Constant propagation with conditional branches. In *ACM Symposium on Principles of Programming Languages*, 1985.