Technical Report

Department of Computer Science
and Engineering
University of Minnesota
4-192 EECS Building
200 Union Street SE
Minneapolis, MN 55455-0159 USA

TR 02-032

Speculative Register Promotion Using Advanced Load Address Table (ALAT)

Tong Chen, Jin Lin, Wei-chung Hsu, and Pen-chung Yew

September 19, 2002

Speculative Register Promotion Using Advanced Load Address Table (ALAT)

Tong Chen, Jin Lin, Wei-Chung Hsu and Pen-Chung Yew
Department of Computer Science and Engineering
Email: {tchen, jin, hsu, yew} @cs.umn.edu

Abstract

The pervasive use of pointers with complicated patterns in C programs often constrains compiler alias analysis to yield conservative register allocation and promotion. Speculative register promotion with hardware support has the potential to more aggressively promote memory references into registers in the presence of aliases. This paper studies the use of the Advanced Load Address Table (ALAT), a data speculation feature defined in the IA-64 architecture, for speculative register promotion. Measurements on SPEC CPU2000 benchmark programs are conducted to show that 1) there is a great potential to reduce the number of load operations using speculative register promotion in several programs; 2) using simple heuristic rules, a compiler can perform alias speculation quite successfully, and can identify the most profitable candidates for speculative register promotion; 3) the execution time of some benchmarks, such as equake, on Itanium machines can be reduced by as much as 17% from the speculation register promotion.

1. Introduction

Register allocation is one of the most important compiler optimizations because the latency of register access is the lowest, and the specification of register is concise. Furthermore, unlike memory references, which may need to go through a hierarchy of caches, the access time of registers is deterministic, so the compiler can conduct more effective instruction scheduling.

In a typical optimizing compiler, register allocation is carried out in two phases: the *register allocation* phase and the *register assignment* phase. In the register allocation phase, the candidate memory references are identified and allocated to an unlimited number of *pseudo registers.* In the register assignment phase, the allocated pseudo registers are mapped to a limited number of physical registers. Many compilers adopted the graph coloring algorithm in the register assignment phase [1, 2, 3].

In the register allocation phase, the compiler identifies as many memory references as possible to be allocated to pseudo registers. In order to be allocated to a pseudo register, a candidate memory reference should not alias with other memory references. A compiler may simply allocate registers based on local information within a statement or a basic block. Such simple register allocation can be improved by allocating scalar variables that have no aliases within a procedure [4]. To further improve register allocation, register promotion techniques [5] are commonly used for potentially aliased memory references. More general register promotion is often applied in the framework of partial redundant elimination [6] to handle control flow structures.

If the compiler has a precise alias analysis phase, many more memory references may be allocated to registers. However, a highly accurate alias analyzer is rather difficult to develop for C programs due to their intensive use of pointers [7]. The imprecise pointer analysis in typical C compilers often results in many possible aliases [8] and prohibits effective register promotion. Furthermore, a compiler must be conservatively correct in register allocation. Even if the probability of a memory reference pair being aliased is very low, the compiler is unable to allocate them to registers. On the other hand, modern processors tend to provide a large register file to allow more memory references to be allocated to registers. It is important to address the disparity.

The alias analysis could be improved, for example, by inter-procedural analysis [9, 10]. However, the extensive use of pointers, especially pointers for dynamically allocated memory objects, requires very powerful inter-procedural alias analysis, which is known to be complicated and expensive [11]. Separate compilation makes inter-procedural alias analysis even more challenging.

One alternative to a more precise alias analysis is to have hardware support for allocating aliased variables to registers. For example, the compiler may speculatively promote possibly aliased memory references into pseudo registers as long as the special hardware can ensure the correctness of data when such ambiguous memory references turn out to be aliased during runtime. Various hardware supports and their respective compiler solutions [12, 13, 14] have been proposed and studied. If there is no special hardware support, the compiler can still speculatively promote possible aliased variables to

registers by generating instructions to check addresses at runtime to ensure the correctness of data [30].

In this paper, we focus on using the Advanced Load Address Table (ALAT), as defined in the IA-64 architectures and implemented in Itanium processors [15], for speculative register promotion. ALAT with the corresponding advanced load and check instructions were originally designed to hide load latency by moving load instructions speculatively ahead of potentially aliased store instructions. In this paper, we discuss the use ALAT to support speculative register promotion and compare this approach with other hardware mechanisms, such as C-regs [13] and SLAT [14], for speculative register promotion. ALAT has the following advantages in register promotion:

- Only store operations need to be checked. In previous designs, such as C-regs and SLAT, all memory operations, including both loads and stores, need to be checked for potential conflicts. The requirement of checking all memory operations could become a performance bottleneck in wide-issue processors.
- The hardware complexity of the ALAT could be lower. The ALAT only needs to detect address conflicts, not to maintain the correctness and consistence of the data stored in registers. If address conflicts occur, load operations are executed to reload latest data into registers. This design can be more cost-effective as long as the speculation is correct most of the time.

However, using ALAT in speculative register promotion has its limitations. It requires all store operations be kept and explicit check instructions be inserted in the code. On the surface, this approach seems not reducing as many instructions as other methods. However, with wide-issue processors, the major performance concern is not on the number of instructions, but the number of expensive operations, such as load operations, in particular, cache-missing loads. Check instructions are not real memory operations, and can be processed like no-ops when the check is successful (i.e. when no conflict is detected)

In this paper, we focus on two key issues related to using ALAT: how to speculatively promote aliased variables to registers, and how to generate correct code to support such promotion. The pros and cons of this approach are also discussed with comparisons to other proposed designs. The effectiveness of speculative register promotion is evaluated with the Intel's ORC compiler [16] on the SPEC CPU2000 benchmarks [17]. The results show that this approach has a good potential to enhance the performance. For example, the measured speedup of one benchmark, *equake*, can be up to 17%.

The major contributions of this paper are:

- A scheme to use ALAT for speculative register promotion. It covers various control structures, such as *if*, *while*, and *function calls,* and address speculations for multi-level pointer variables. This scheme is able to speculatively promote not only scalar variables but also indirect memory references such as pointers, which were not attempted in most of other schemes.
- Experimental results that show the potential of speculative register promotion and more detailed classifications provide us insights on which techniques are more performance critical.

The rest of this paper is organized as follows. The concept and the structure of ALAT are described briefly in the next section. The method that applies the ALAT to speculative register promotion is presented in section 3. Section 4 evaluates the effectiveness and efficiency of our approach. The advantages of using ALAT are compared with other related works in section 5. Finally, we draw our conclusions in section 6.

2. Advanced Load Address Table (ALAT)

ALAT, originated from the memory buffer concept [31], is designed to support data speculation in code scheduling. With ALAT, load instructions can be speculatively moved ahead of potential aliased stores to hide the load latency. Both the Itanium and Itanium-2 processor have implemented ALAT. We briefly describe the functionality of the ALAT based on the Itanium implementation [15].

ALAT is illustrated in Figure 1. Each entry in this table records information of a speculative load. When a speculative load instruction *ld.a* is issued, an entry in ALAT is allocated to record the target register number, a partial memory address, and the size of the data. An old entry may be evicted to make room for the new one. Every store operation needs to check its store address against all of the addresses recorded in ALAT. If there is a match, the corresponding entry of ALAT is invalidated. The case of an address match is called a collision.

A check is performed by *ld.c* or *chk.a* before speculatively loaded data is used. If a valid entry for the advanced load indexed by the target register number is present in ALAT at the time of the check, no conflicts have occurred. The data in the target register is considered valid and can be directly used. Otherwise, the check fails, and the correct data must be reloaded. The *ld.c* instruction simply reloads the data from memory. The *chk.a* instruction will jump to a recovery routine specified in the check instruction. The *chk.a* instruction provides more flexibility for recovery because the instruction scheduler may move the load instruction, as well as some subsequent data-dependent instructions,

across potential aliased stores. If subsequent data-dependent instructions are also moved speculatively, such operations must be re-computed in the recovery routine when the check fails. The overhead of the recovery routine may be very high if the code scheduling is too aggressive. After the check is performed, the corresponding entry in ALAT can be either kept or cleared, depending on the clear or non-clear completer specified in the check instructions. An entry can also be explicitly cleared by the invalidation instruction, *invalva*.

reg	address	size

a. structure of ALAT

```
ld.a r12=[r11]

st   [r13]=r14
ld.c r12=[r11]
```

When the addresses held in r11 and r13 are possible aliases at compiler time, advanced load and check can help move the load ahead of the write.

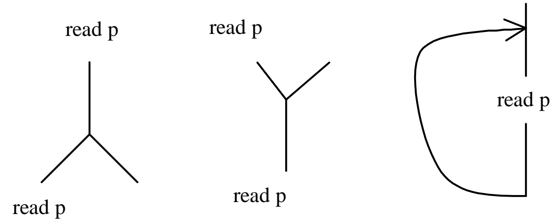
b. an example of advanced loads

Figure 1: The advanced load address table for data speculation

3. Speculative Register Promotion

3.1 Overview

In a traditional non-speculative register promotion scheme, redundant loads for one memory reference within a region are identified. The data item of such references is kept in a register so that the redundant load operations can be removed. It is similar to allocating this data item to a register. A load operation may be redundant when it is the second load in a read-after-read (RAR) dependence or a read-after-write (RAW) dependence. When control flow is considered, redundant load operations may appear in three different cases: total redundancy, partial redundancy and redundancy in the form of a loop-invariant. The three possible cases with RAR dependency are illustrated in Figure 2. Algorithms have been proposed to handle these cases in traditional non-speculative register promotion schemes [6].



a. total redundancy   b. partial redundancy   c. redundancy of a loop invariant

Figure 2: Redundancy cases with control flow

The speculative register promotion adds alias speculation to the traditional register promotion. A data item may be speculatively kept in a register to avoid redundant load operations even when there are potential aliased stores exist. When a possible aliased store turns out to be true alias at runtime (called a *collision*), the collision has to be detected and the correct value of registers reloaded. We use ALAT to serve this purpose. The key issues here are how to speculatively promote references into registers and how to generate correct code. These two issues are discussed in the following sections.

3.2 Speculative Data Flow Analysis

In a typical compiler, register promotion is performed based on the results of data flow analysis, usually in the Static Single Assignment (SSA) form [6]. In turn, data flow analysis is performed based on the results of pointer analysis. The impreciseness of alias analysis is usually the main reason that register promotion is not very effective. *Alias speculation* is one way to increase the effectiveness of register promotion.

In alias speculation, we distinguish highly probable alias pairs from low probable alias pairs. The probability of aliases can be estimated with either heuristic rules or runtime profiling information. We describe a simple speculative data flow analysis that allows us to compute speculative alias information in this section.

The aliases determined by a traditional compiler analysis are usually divided into three categories: *definite aliases*, *no aliases* and *may aliases*. The first two categories are precise while the last one is rather conservative. Our approach is to initially keep the definite aliases and no aliases computed by the compiler, and optimistically treat all the may aliases as no aliases. Then, a data flow analysis is performed based these assumptions. Due to such speculation, many weak updates originally caused by may aliases are now removed. Consequently, more definite aliases can be derived, and thus more candidates for speculative register promotion.

```
Initialize definite_alias set to be the result of traditional pointer
analysis and initialize may-alias set to be empty.
Do {
        Data flow analysis based on the definite_alias set and
may_alias set;
        switch (the address  parts of two memory expressions,
e1 and e2) {
        case (if they have the same value along all control flow
paths): add (e1, e2) to the definite_alias set;
        case (if they can be found to have the same value along
at least one of the control flow paths, but not all): add (e1, e2) to
the may_alias set;
        otherwise: do nothing;
        }
} until { definite_alias set and may_alias set do not change)
```

Figure 3: Algorithm for speculative data flow analysis

The speculatively derived definite aliases are determined by a data flow analysis and a heuristic rule. The heuristic rule we use is very simple. It assumes that only references with the same expression form can be aliases. The data flow analysis is to determine whether two references have the same definition points or same version number, as discussed in [23]. References having the same expression and having the same definition points for each variable in the expression are considered definite aliases. A backward lookup [20] can be used to determine the impact of intermediate temporary variables. If our speculative alias analysis is too aggressive, it may promote too many non-profitable references which will degrade the overall performance. Therefore, in addition to the definite aliases, our compiler also generates some may aliases guided by the results of data flow analysis. A high level description of the speculative data flow analysis algorithm is presented in Figure 3. Since the conventional points-to analysis is performed first, our compiler can tell where speculation occurs.

3.3 Code Generation for Speculative Register Promotion

With the speculative alias sets obtained from a speculative data flow analysis, we then use traditional partial redundancy elimination algorithms [6] to identify redundant load operations. Its details are beyond the scope of this paper.

The identified redundant loads can be total redundant, partial redundant or as loop invariants as shown in Figure 2. Redundant loads can be removed. The remaining issue is how to treat those redundant references that are speculative. The following sections describe how to insert advanced loads and runtime checks for those speculative redundant references.

3.3.1 Basic transformations

We start with the simplest case: redundant loads with total redundancy, as shown in the example in Figure 2(a). Basic transformations are as follows:

- The first load is replaced by an advanced load, *ld.a,* so that an entry can be allocated in ALAT
- The second load is replaced by a check (*ld.c* or *chk.a*). If there is no conflict detected by the check, i.e., there is no aliased stores between the advanced load and the check, *ld.c* is simply executed as a no-op. If there is a conflict, the check instruction will cause the up-to-date data loaded from memory.

The transformation for RAR and RAW differ only in the handling of the leading reference operation. When the leading reference operation is a read, the leading load is replaced by *ld.a*. When the leading reference operation is a write, a *ld.a* instruction is added after the store instruction to secure an entry in ALAT. When there are more than two reads to the same register, each read in the middle of the sequence should use a check with the non-clear completer, for example, *ld.c.nc*, so that the entry can remain in ALAT after each check. An example of three read references is shown in Figure 4.

original code	after transformation
=p+1;	ld.a r1=[p] add r3=r1,1
=p+3;	ld.c.nc r1=[p] add r4=r1, 3
=p-5;	ld.c r1=[p] sub r5=r1, 5

Figure 4: An example of multiple references

3.3.2 Transformations with control flows

For partially redundant loads, such as the second load in Figure 2(b), it is not always beneficial to eliminate them because extra load instructions may be needed to cover all control flow paths [9]. To avoid performance degradation, the transformation is often guided by certain heuristic rules or branch profiling information. The same approach can be applied to speculative register promotion. There is an instruction, *invala,* to invalidate a single entry of ALAT. This instruction can be inserted to a dominating point to handle partial redundancy, as shown in Figure 5. The invalidation instruction is not a memory operation, so it is cheaper than a load

instruction. Since no memory address is specified in the invalidation instruction, there are no data dependences involved in this instruction. This means the invalidation instruction is likely to be scheduled for free. The disadvantage of using the invalidation instruction is that it may increase the life time of a register. However, with a large register file in Itanium, the register pressure is usually not a big problem.

a. Original Code	b. Speculative register promotion.

if ( ) {
  =p+1
}
 	
if () {
=p+3
}
	invala.e r1
	if() {
	ld.a r1=p
	add r3=r1, 1
	}
	
	if() {
	ld.c r1=p
	add r4=r1, 3
	}

Figure 5: An example of partial redundancy

Original	Promoted

while () {
 	
  =p+1
}
 	
There is a possible alias write in the loop that may modify p.
	ld.sa r1=[p]
	
	while () {
	
	  chk.a.nc r1=[p]
	  add r3=r1, 1
	
	}

Figure 6: An example of loop

When data is reused across an entire loop (see Figure 6), the load operation can be moved speculatively out of the loop. In speculative register promotion, such a load is not only control speculative, but also data speculative. The instruction *ld.sa* in IA-64 should be used here. We only need one check instruction, *chk.a*, to check both control and data speculation. The check instruction should keep the entry in ALAT (i.e. the *chk.a* flag should be set to not-clear) because each of the subsequent iterations needs to use the allocated ALAT entry. Figure 6 shows the code to speculatively promote a speculative loop invariant to a register.

3.3.3 Procedure calls

Possible aliased store operations may occur within procedure calls. Again, we can use ALAT to support speculative register promotion across procedure calls.

When a procedure is called, all store operations in the called procedure will automatically check against the ALAT. Collisions caused in the callee can be detected after the call returns. However, it is possible that the same register may be re-allocated in the callee because different procedures can use the same register. Thus, when a called procedure returns, all ALAT entries allocated in the procedure have to be cleared. Otherwise, the ALAT checks after the procedure call may cause errors because conflict entries may have been unexpectedly replaced due to register re-allocation.

For example, assume *r11* is allocated in ALAT by procedure *P*, and this entry is invalidated by some store operation in procedure *P*. Assume procedure *P* calls procedure *Q*. In procedure Q, *r11* is a re-allocated entry in ALAT. If this entry is not cleared before procedure Q returns, the checks in procedure *P* after the return from Q would find a valid entry in ALAT. Such errors could be avoided by invalidating all entries that are allocated in procedure *Q*.

There are two ways to invalidate entries in ALAT. If the last check of the entry can be identified, the last check should have a clear completer, i.e. *ld.c.clr*, to invalidate the entry. If the entry is allocated by a called procedure, it can be explicitly removed using the instruction *invala.*

3.3.4 Cascade failure

When a pointer reference and the data it points to are both speculatively promoted to registers, a collision detected by the check of the pointer reference will cause both the pointer and the data it points to be reloaded. This is called a cascade failure [32]. Such cases may happen when the address part of the reference may have aliased writes. For instance, **p*, *a[j]* and ***q* may have aliases in their address part if p and q are global variables, or if their addresses have been taken. The check instruction, *chk.a,* can be used to handle cascade failures. When the advanced load check fails, the instruction *chk.a* will jump to a recovery routine. In the recovery routine, both the address and the data can be reloaded. Figure 7(a) shows such an example. Figure 7(b) shows the transformation when only the address may be modified, and Figure 7(c) shows the transformation when both the address and the data may be modified. All previous discussions on speculative register promotions are applicable to pointer references. The main difference is that the check instruction should be *chk.a*, instead of *ld.c.*

3.4 Overhead for Speculative Register Promotion

When the leading reference is a read, there is no overhead because the original *ld* instruction is replaced by *ld.a*. When the leading reference is a write, an additional instruction, *ld.a,* is inserted after the store operation. With a minor modification to the hardware, this operation could be combined with the store instruction to save an extra *ld.a* instruction. For example, we can define a new *st.a* instruction. Like the *ld.a* instruction, a *st.a* will allocate an entry in ALAT.

a) source code	b) p, the address of *p, may be modified	c) both p and *p may be modified
= *p+1	ld.a r1=p ld r2=[r1]	ld.a r1=p ld.a r2=[r1]
…	… chk.a r1, #recovery add r3=r2, 3	… chk.a r1, #recovery ld.c r2=[r1] add r3=r2, 3
= *p+3	#recovery: ld r1=p ld r2=[r1]	#recovery: ld r1=p ld.a r2=[r1]

Figure 7: Examples of cascade failure

The following reads must be checked with *ld.c* or *chk.*a. The *ld.*c can be executed concurrently with the instructions that use its result, and takes zero cycle if there is no collision. The *chk.a* may not be scheduled in the same bundle as the consumer. However, its recovery scheme can support more aggressive speculative code scheduling.

A check instruction will incur no overhead if there is no check failure and there is a free slot to schedule it. If the check fails, a *ld.c* will simply reloads the data from the memory. Nevertheless, the load latency is now exposed. For the *chk.a*, there is a relatively large penalty to jump to and back from the recovery code. This penalty includes a light-weighted trap and a branch mis-prediction. Therefore, a mis-speculation, especially for address mis-speculation, could be very expensive.

When registers are speculatively promoted cross procedure calls, extra *invala* instructions operation may be needed in the called procedure. In some cases, the invalidation instructions can be combined with the check instructions and incur no cost. Since the *invala* instruction is a not a memory instruction, and does not have true data dependency, it is likely to be scheduled for free.

4. Evaluation of Speculative Register Promotion Using ALAT

The following aspects of speculative register promotion are evaluated:

- Impact on the reduction of load instructions. The number of load operations that can be potentially reduced is estimated by a trace analysis. The baseline used is Intel's Open Research Compiler (ORC). There are many state-of-art optimizations implemented in this compiler.
- Effectiveness of speculation. The algorithm of a speculative data flow analysis (discussed in section 3.2) is simulated. The candidates for speculative register promotion that can be identified by this algorithm are counted. The data shows the alias speculation using a speculative data flow analysis can identify most of the profitable candidates for speculative register promotion.
- Impact to performance. One benchmark, equake, is optimized manually with speculative register promotion, and the performance improvement is measured on an Itanium machine. For all the other benchmarks, the mis-speculation rates are collected and reported so that the performance improvements can be estimated.

4.1 Impact on Load Instruction Reduction

The main objective of speculative register promotion is to reduce load instructions. We measure the number of loads that can potentially be reduced by speculative register promotion after the ORC compiler has applied all its analyses and optimizations.

The most relevant analysis and optimizations in the ORC compiler are pointer analysis, register promotion and partial redundancy elimination. The pointer analysis in the ORC compiler begins with flow-free analysis [24], and further improved by flow sensitive analysis [9, 25]. The register promotion and partial redundancy elimination are performed at the same time [6]. The ORC compiler is able to promote scalar references as well as indirect references to registers. Current version of our ORC compiler does not perform inter-procedural alias analysis (next version of the ORC compiler will).

Table 1 reports the number of potential load operations that can be further reduced by speculative register promotion. Both of the static and dynamic numbers are reported. The first two data columns report the potential when the side effects of function calls are assumed unknown and redundant load operations can only be identified within functions. The last two columns report the potential when the side effects of function calls are known and register promotion can cross function boundaries.

Table 1: Potential of speculative register promotion

	Within function calls		Across function calls	
	static load reduced	dynamic load reduced	static load reduced	dynamic load reduced
ammp	24.3%	49.9%	25.0%	50.3%
art	38.5%	64.2%	38.5%	64.2%
equake	31.6%	47.3%	31.6	47.3%
bzip2	20.4%	5.6%	20.4	5.6%
crafty	9.5%	8.1%	10.2%	11.4%
gap	25.2%	35.8%	25.2%	35.8%
mcf	22.0%	10.8%	22.0%	10.8%
parser	5.3%	5.9%	5.8%	6.1%
twolf	31.0%	29.9%	31.0%	29.9%

The data shows that there is great potential for speculative register promotion, especially for floating point benchmarks. An example extracted from the procedure *smvp* of the *equake* benchmark is given in Figure 8. This procedure takes nearly 60% of the total execution time. There are many similar references in the loop. We show only selected parts of the two statements in this example. In the example, there are three load operations, A[Anext], A[Anext][0] and A[Anext][0][1] for the array reference A[Anext][0][1] (in bold font) since A is a three-level pointer. These references are not allocated to registers because w[col][0] is a possible aliased write. All these pointers point to the heap memory. However, the first two load operations, A[Anext] and A[Anext][0] can be speculatively promoted into registers. Many references, such as v[i], v[i][0], and w[col] are also candidates for promotion. As a result, 39.8% of all load operations in this benchmark can be removed. Of course, these load operations can also be reduced If the compiler can have precise inter-procedural pointer analysis. However, a simple local analysis plus speculative register promotion may be more practical and cost effective.

Table 1 also shows that the register promotion across function calls does not provide significant performance gain in most of the benchmarks. One reason may be that we assume perfect control speculation in the measurements, and function calls not in the executed paths were not considered. The results of table 1 seems to indicate that the side effect analysis for function calls may not be needed for speculative register promotion. Therefore, the next experiments focus more on register promotion within functions.

```
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];
      . . .
     w[col][0] += A[Anext][0][0]*v[i][0] + . . .;
     w[col][1] += A[Anext][0][1]*v[i][0] + . . .;
      . . .
     Anext++;
    }
   . . .
 }
}
```

Figure 8: An example of speculative register promotion in the benchmark equake

4.2 Effectiveness of Speculative Data Flow Analysis

In the section, we present the results of the simple speculative data flow analysis discussed in section 3. We use the ORC compiler to instrument all assignments and memory references in a program to generate memory reference traces, and we simulate the algorithm using the generated memory reference traces. The formats of each memory reference are compared and classified by the instrumentation tool.

Table 2 reports the results of speculative register promotion based on our algorithm. Since the results in Table 1 show no significant advantages on promoting registers speculatively across function calls, we only consider speculative register promotion within functions. The first column in Table 2 shows the percentage of load reduced. Additional information is also provided for the reduced loads. For example, the second column reports how much reduced loads are due to register promotion of indirect references. The third column reports how much loads reduced are due to promotion with possible cascaded failures (as described in section 3.3.4). The last column reports how much reduced loads are due to RAR dependences (as described in section 3.3.1), as oppose to RAW dependences. The three categories may overlap.

Table 2: Speculative register promotion

	load reduced	Among all the reduced load operations		
		Indirect references	Possible Cascade failure	RAR dependence s
ammp	49.5%	100%	0%	53.2%
art	59.3%	10.5%	2.1%	89.4%
equake	39.8%	81.7%	38.3%	98.7%
bzip2	5.0%	6.0%	0.8%	96%
crafty	8.8%	19.0%	0%	55.2%

gap	23.3%	66.9%	12.8%	91.7%
mcf	3.5%	100%	2.5%	88.6%
parser	5.6%	30.0%	0%	37.2%
twolf	27.2%	27.3%	13.6%	76.8%
Average	26.3%	51.2%	7%	76.3%

Compared with Table1, Table 2 shows that the speculative data flow analysis may effectively identify potential candidates for speculative register promotion. Among all the removed loads, over 50% are from indirect references. Previous schemes [13] that speculatively promote only named scalars would miss this portion of the performance improvement. 7% of the removed loads come from cascaded references. Our approach can handle this portion of references reasonably well. However, promote these references may have the risk of causing cascade failure (see section 3.3.4), and incur a high recovery cost. Hence, they should be handled more selectively. The loads reduced due to RAR dependence are about 76.3%. This is in line with our expectation. So the need for a new instruction, *st.a*, may not be critical.

4.3 Performance

The performance of speculative register promotion is largely determined by the mis-speculation rate. Since the penalty for mis-speculating the data and the address is quite different, the rates of failure for these two kinds of speculation are reported separately in Table 3. The rate is the number of times that collisions occur over the number of load operations that are selected as candidates for speculative register promotion. Both mis-speculative rates seem to be low enough to justify the proposed speculative register promotion.

Table 3: Measurement of mis-speculation rates

	mis-speculative rate of speculative register promotion	mis-speculative rate of cascade failure
ammp	0.01%	0.002%
art	1.5%	0.0%
equake	0.02%	0.0%
bzip2	0.09%	0.0%
crafty	0.2%	0.07%
gap	1.8%	0.38%
mcf	0.0%	0.0%
parser	0.1%	0.006%
twolf	0.002%	0.0%

We further evaluate the performance of speculative register promotion by running a manually compiled benchmark on an Itanium machine (a 733 Mhz HP i2000 workstation). The *equake* benchmark is selected because it has a high potential for speculative register promotion. The procedure *smvp* takes up 56% of the total execution time of equake. By speculatively promoting frequently referenced memory expressions to registers in this procedure, we obtain 17% overall performance improvement with train input.

One important lesson learned in optimizing this procedure is that conflicts in ALAT due to registers must be minimized. Remember that ALAT entries are indexed by the physical register number specified in the *ld.a/ld.c* instructions. Since the advanced loads need a large number of registers, careful register assignment can avoid conflicts in ALAT: the ALAT has 32 2-way associative entries. By promoting the address computation into registers in this procedure, it can further reduce the load operations for floating point data. All these techniques together can achieve significant performance improvements.

5. Related Work and Discussions

Several hardware designs have been suggested to support speculative register promotion. Ben. Heggy and Mary Lou Soffa [12] presented a hardware design that maintains the data coherence between registers and memory. The address of each memory access is checked and the access may be redirected to registers. Changes of registers are forwarded to registers in the forwarding group. C-reg [13] is another design that simplifies the register forwarding. The Store-Load Address Table (SLAT) [14] is designed to support speculative register promotion. A separate table records the addresses of data that have been allocated to registers. The ALAT can be used for speculative register promotion. Partial implementation can be found in Intel's Electron compiler [33] and HP's Itanium compiler. No discussion and evaluation on the partial implementations have been published. We will compare the ALAT approach with other proposals and discuss some related issues.

5.1 Reducing Redundant Loads and Stores

In this paper, we focus more on removing redundant loads while other works [12, 13, 14] can remove both redundant loads and stores. Although it is important to eliminate store operations because typical processors may not handle as many stores as loads. However, we believe it is more important to reduce load operations for modern and future processors. Load operations usually are the leading instructions on data dependence graphs, the load latency often increases the critical path length. The ALAT approach basically replaces loads by the advanced load check, such as *ld.c,* which incur no latency as long as the speculation succeeds. One trend of recent processors is to have a wider issue capability and a faster clock rate. It is getting more difficult to design

the first level data cache to achieve a 1-cycle hit latency. Therefore, the main contribution of speculative register allocation would be the reduction of latency rather than reducing the number of instruction executed.

Furthermore, using ALAT for speculative register promotion, only store operations have to be checked. Other proposals require all memory operations to be checked at runtime, which may become a bottleneck for wide issue implementations.

5.2 Coherence Maintenance vs. Recovery

Previous hardware support proposals [12, 13, 14] for speculative register promotion maintain correct data value for each register when aliased writes occur. The latest data value is either forwarded to relevant registers or update in registers. The ALAT adopts a simpler approach that remembers an occurrence of collision, and relies on recovery to obtain the correct data value. The ALAT approach has a higher mis-speculation penalty, but it has clear advantages when the speculation is right:

- A smaller table size. Since nowadays processors have a large register file, the table to maintain the data coherence needs to be as large as the register file. Otherwise, the hardware must handle complicated table spills. The ALAT in Itanium processors actually use a smaller table. Unlike other mapping tables that maintain data coherence between register and memory, ALAT does not need to deal with spilling since conflicts in ALAT will simply get discarded, and will not cause any correctness issues. Excessive conflicts will trigger more check failures to issue reloads from memory.
- ALAT is not part of the architecture states. The ALAT does not have to be stored at context switching, or procedure calls. In other designs, special stores and recovery are required.
- Full address vs. partial address. Unlike other mapping tables, ALAT does not need to match addresses exactly. Even if ALAT treats two different addresses as the same address and mistakenly revoke one allocated entry, it does not incur any correctness issues. In this case, only additional check failures will be triggered, and more reloads to be issued. For this reason, ALAT can afford to use partial address (for example,. 12 to 20 bits) rather than the full address in the table. This is important for processors with very high clock rate because a full address comparison may not be complete within a clock cycle.

As long as the compiler is able to speculate effectively, the recovery approach adopted by ALAT can be more cost effective than that to maintain the coherence of registers.

5.3 A Software-only Approach

The alias problem in speculative register promotion can also be solved by pure software approach [30] as illustrated in Figure 9. This approach has been implemented in the ORC compiler. When two references may be aliased, their address values are compared and a predicated instruction for register forwarding is added. This transformation also avoids redundant load operations when aliased writes occur at runtime. Redundant loads are removed at the cost of a comparison and a conditional register copy instruction.

The major advantage of using ALAT is that the comparison of addresses is done implicitly in hardware. The software approach has to generate an explicit compare instruction for every possible alias reference after every aliased store operation. If there are multiple possible aliased writes in between, there will be many compare instructions and conditional register copy instructions, which could slow down the program. On the other hand, the ALAT performs comparison for all aliases at the same time for the store operation. Two approaches have different impact on code scheduling. The check instruction, *ld.c*, can be scheduled in the same bundle of the instruction that uses the register. So the length of dependence chain is shorter using the ALAT approach. An advantage of the software approach is that only arithmetic operations, instead of memory operation, are used, and therefore, there is no mis-speculation penalty.

```
= *p        ld r1=[r2]  //r2 holds the value of p;
                        //r1 holds the value of *p;
*q=         st [r3]=r4  //r3 holds the value of q
                        //r4 holds the new value of *q
= *p        comp.eq.unc p1, p0=r3, r2
                        //test whether *p and *q are aliases
            (p1) mv r1=r4  //if yes, update r1 with r4
                        //anyway, *p is not reloaded.
```

Figure 9: Software approach for alias problem

5.4 Interaction with Advanced Load Scheduling

Our speculative register promotion approach does not conflict with the original design objective of ALAT, i.e. to hide the latency of loads. The advanced loads are intended to speculatively move individual loads as early as possible [15]. When the advanced loads are used for speculative register promotion, groups of references are handled together. Therefore, not only the latency of loads is hidden, but also redundant loads are removed. After the speculative register promotion, the scheduler could still use the advanced loads to hide the latency of the loads for the first producer references.

6. Conclusions

Register allocation is an important optimization in a compiler. The effectiveness of register allocation is often limited by imprecise alias analysis in C compilers due to intensive use of pointers. Although much progress have been made on improving pointer analysis in C compilers, analyzing heap-oriented pointers remains a challenge to current compilers. Speculative register promotion is a technique to assist register allocation when the compiler fails to allocate memory objects with unresolved aliases to registers. This paper examines the use of the Advanced Load Address Table (ALAT), defined in the IA-64 architecture, to perform speculative register allocation.

The code generation issues of speculative register promotion using ALAT are discussed. Examples on how to perform speculative register promotion are provided for basic blocks, various control flow structures, and indirect reference chains. A speculative data flow analysis scheme based on simple heuristics is proposed for the compiler to identify candidates for register promotion.

The Intel ORC compiler is used to evaluate the performance potential of speculative register promotion using ALAT. The ORC compiler is modified to instrument SPEC2000 benchmarks to generate memory traces with annotations. Using trace analysis, the potential of redundant load elimination from speculative register promotion is reported. The proposed speculative data flow analysis for speculative register promotion is simulated to evaluate its performance. In the simulation, the mis-speculation rate for each memory object speculatively promoted is also reported.

The analysis and experiments show that the proposed scheme is quite effective. It achieves most of the potential of redundant load elimination. On average, 26% additional load operations may be reduced for a set of SPEC2000 programs optimized by the ORC compiler. In addition to scalars [14] or array elements [13], our scheme promotes indirect references with multiple-level pointers, which are increasingly important in modern C programs. Our simulations also show that speculatively promoted memory references have a very low mis-speculation rate, which means recovery penalty will be low. One Spec2000 benchmark, equake, has been optimized based on the proposed scheme, and achieves 17% of speed up on a HP i2000 Itanium workstation. The integration of this scheme with other optimization phases in the ORC compiler is currently under development.

References

[1] G.J. Chaitin. Register allocation and spilling via graph coloring. In Proceedings of the ACM SIGPLAN 82 Symposium on Compiler Construction, pages 98-105, New York, NY, 1982. ACM.

[2] P. Briggs, K. Cooper, K. Kennedy, and L. Torczon. Coloring heuristics for register allocation. In ASCM Conference on Program Language Design and Implementation, pages 275-284, 1989.

[3] Fred C. Chow and John L. Hennessy. The priority-based coloring approach to register allocation. ACM Transactions on Programming Languages and Systems, 12(4):501-536, 1990.

[4] D. W. Wall. Global register allocation at link time. In Proceedings of the ACM SIGPLAN '86 Conference on Programming Language Design and Implementation, pages 264--75, 1986.

[5] K. D. Cooper and J. Lu. Register Promotion in C Programs. In Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation, pages 308--319, Las Vegas, NV, June 1997.

[6] Raymond Lo, Fred Chow, Robert Kennedy, Shin-Ming Liu, and Peng Tu. Register promotion by sparse partial redundancy elimination of loads and stores. In Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI), pages 26--37, Montreal, Canada, 17--19 June 1998.

[7] Tong Chen, Jin Lin, Wei-Chung Hsu and Pen-Chung Yew, On the Impact of Naming Methods for Heap-Oriented Pointers in C Programs, International Symposium on Parallel Architectures, Algorithms, and Networks, 2002

[8] Tong Chen, Jin Lin, Wei-Chung Hsu and Pen-Chung Yew, "The Empirical Study of the Granularity of Pointer Analysis in C programs. LCPC 20002.

[9] Choi, M. Burke, and P. arini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and sife-effects. In Proceedings of the ACM 20th Symposium on Principles of Programming Languages, pages 232-245, January 1993.

[10] W. Landi and B.G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In proceedings of the SIGPLAN'92 Conference on Programming Language Design and Implementation, page 235-248, July 1992.

[11] Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for C programs. In Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation, pages 1-12, June 1995.

[12] B. Heggy and M. L. Soffa. Architectural Support for Register Allocation in the Presence of Aliasing. Proc., Supercomputing '90: November 12-16.

[13] H. Dietz and C.-H. Chi. CRegs: A New Kind of Memory for Referencing Arrays and Pointers. Proc., Supercomputing '88: November 14-18.

[14] Matthew Postiff, David Greene, Greene and Trevor Mudge. The Store-Load Address Table and Speculative Register Promotion. Proc.33rd Annual Intl. Symp. Microarchitecture (Micro33), Monterrey, CA. December 10-13, 2000, pp. 235-244.

[15] Intel software college: http://developer.intel.com/software/products/college/itanium/

[16] Roy Ju, Sun Chan, and Chengyong Wu. Open Research Compiler for the Itanium Family. Tutorial at the 34th Annual International Symposium on Microarchitecture.

[17] Spec CPU2000, http://www.specbench.org/osg/cpu2000/

[18] Peter Dahl and Matthew O'Keefe. Reducing Memory Traffic with CRegs. Proc. 27th Intl. Symp. Microarchitecture, pp. 100-104, Nov, 1994.

[19] M. Emami, R. Ghiya, and L. J. Hendren. Context-Sensitive Interprocedural Points-to Analysis in the Presence of Function Pointers. In Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation, pages 242-256, June 1994.

[20] Eric Stoltz, Michael Wolfe, and Michael P. Gerlek. Constant propagation: A fresh demand-driven look. In ACM Symposium on Applied Computing, Phoenix, Arizona, March 1994. ACM SIGAPP.

[21] R. D.-C. Ju, K. Nomura, U. Mahadevan, and L.-C. Wu, "A Unified Compiler Framework for Control and Data Speculation," Proc. of 2000 Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT), pp. 157 - 168, Oct. 2000.

[22] R. D.-C. Ju, J.-F Collard, and K. Oukbir, "Probabilistic Memory Disambiguation and its Application to Data Speculation." In Proc. Of the 3rd Workshop on Interaction between Compilers and Computer Architectures, Oct. 1998.

[23] Fred Chow, Raymond Lo, Shin-Ming Liu, Sun Chan, and Mark Streich, Effective Representation of Aliases and Indirect Memory Operations in SSA Form, Proc. of 6th Int'l Conf. on Compiler Construction, April 1996, pp. 253-257

[24] Bjarne Steensgaard. Points-to analysis in almost linear time. In Conference Record of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages, Pages 32-41, January, 1996.

[25] W. Landi and B.G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In proceedings of the SIGPLAN'92 Conference on Programming Language Design and Implementation, page 235-248, July 1992.

[26] L.Carter, B.Simon, B.Calder, L.Carter, and J.Ferrante. Predicated static single assignment. In Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, October 1999.

[27] Rakesh Ghiya, Daniel Lavery and David Sehr. On the Importance of Points-To Analysis and Other Memory Disambiguation methods For C programs. In Proceedings of the ACM SIGPLAN'01 Conference on Programming Language Design and Implementation, page 47-58, June 2001.

[28] Eric Stoltz, Michael Wolfe, and Michael P. Gerlek. Constant propagation: A fresh demand-driven look. In ACM Symposium on Applied Computing, Phoenix, Arizona, March 1994. ACM SIGAPP.

[29] 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 SIGPLAN '97 Conference on Programming Language Design and Implementation (PLDI), pages 273--286, Las Vegas, Nevada, May 1997.

[30] A. Nicolau. Run-time disambiguation: Coping with statically unpredictable dependencies. IEEE Transactions on Computers, 38(5):663--678, 1989.

[31] D. M. Gallagher, W. Y. Chen, S. A. Mahlke, J. C. Gyllenhaal, and W. W. Hwu. Dynamic memory disambiguation using the memory conflict buffer. In 6th International Conference on Architectural Support for Programming Languages and Operating S ystems, pages 183-193, Oct. 1994.

[32] R. D.-C. Ju, K. Nomura, U. Mahadevan, and L.-C. Wu, "A Unified Compiler Framework for Control and Data Speculation," Proc. of 2000 Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT), pp. 157 - 168, Oct. 2000.

[33] R. Krishnaiyer, D. Kulkarni, D. Lavery, W. Li, C.-C. Lim, J. Ng, D. Sehr, An advanced optimizer for the IA-64 architecture, IEEE Micro, Vol. 20, No. 6, Nov./Dec. 2000.