Speculative Register Promotion Using Advanced Load Address Table (ALAT)

Jin Lin, Tong Chen, Wei-Chung Hsu and Pen-Chung Yew
Department of Computer Science and Engineering
University of Minnesota
Email: {jin, tchen, 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. An algorithm for speculative register promotion based on partial redundancy elimination is presented. The algorithm is implemented in Intel's Open Research Compiler (ORC). Experiments on SPEC CPU2000 benchmark programs are conducted to show that speculative register promotion can improve performance of some benchmarks by 1% to 7%.

1. Introduction

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 be aliased 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, many more memory references may be allocated to registers. Although the static pointer analysis has made significant progress in recent year, a highly accurate alias analyzer is still 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 still cannot 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, 11, 27]. However, the extensive use of pointers with complex patterns, especially pointers for dynamically allocated memory objects, requires comprehensive inter-procedural alias analysis, which is known to be complicated and expensive [11]. Separate compilation and the extensive use of shared library makes inter-procedural alias analysis even more challenging.

One alternative to a more precise alias analysis is to have hardware support for allocating aliased memory references 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, 18] 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], to help 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. 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 ALAT only needs to detect address conflicts, not to maintain the correctness and consistence of the data stored in registers, and, thus, the hardware complexity of the ALAT is much lower. If address conflicts occur, load operations are executed to reload latest data into registers. This design can be very 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 does not appear that it can reduce 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: the scheme to use ALAT to help speculative register promotion, and the algorithm to perform such promotion in a compiler. 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.

The major contributions of this paper are:

- A scheme to use ALAT for speculative register promotion. 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 [14].
- An algorithm for speculative register promotion based on partial redundancy elimination (PRE). In addition to control speculation, this algorithm further introduces alias speculation into register promotion.
- Implementation in the ORC compiler. We implemented our algorithm in the ORC compiler. The experimental results show that the execution time of some SPEC CPU2000 benchmarks on Itanium machines can be reduced by 1% to 7%.

The rest of this paper is organized as follows. The scheme that applies the ALAT to speculative register promotion is described in section 2. Section 3 discusses the compiler algorithms for speculative register promotion. We report our experiment results in section 4. Related works are compared in section 5. Finally, we draw our conclusions in section 6.

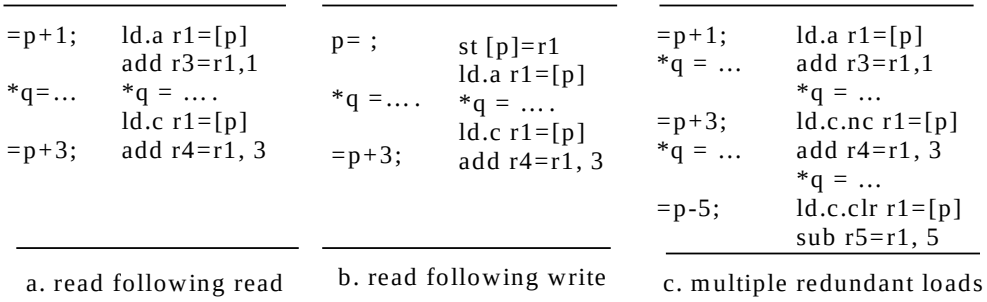
2. Schemes for speculative register promotion

2.1. Advanced load address table (ALAT)

ALAT, originated from the memory buffer concept [31], is designed to support data speculation in code scheduling. Both the Itanium and Itanium-2 processor have implemented ALAT. We briefly describe the functionality of the ALAT based on the Itanium implementation [15].

When a speculative load is issued with a special load instruction, *ld.a*, the target register number, the partial memory address, and the size of the data are stored in an entry for this speculative load. Every store operation automatically compares 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*.



a. read following read b. read following write c. multiple redundant loads

Figure 1. Examples of basic transformations

2.2. Basic transformations

We start the description of the scheme for speculative register promotion with the simplest case: a redundant load following a load operation. Basic transformations are as follows (Figure 1 (a)):

- The first load is replaced by an advanced load instruction, *ld.a*.
- The second load is replaced by a check instruction, *ld.c*.

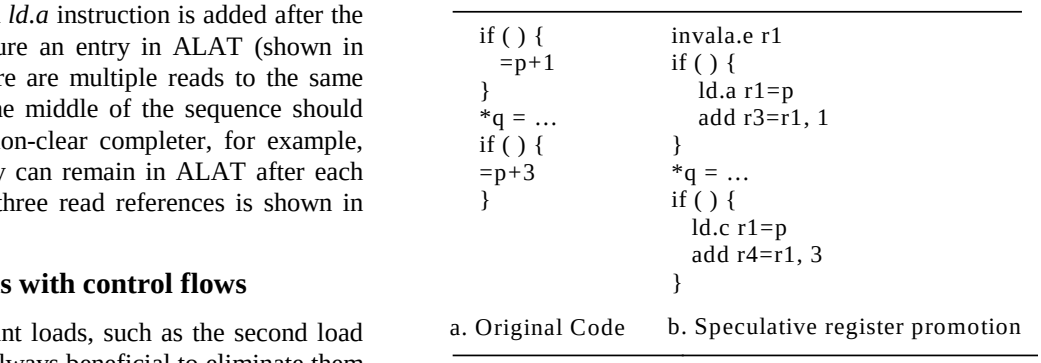
The *ld.a* allocates an entry in ALAT. If there is no conflict detected by the check, i.e., there is no aliased stores occurred between the advanced load and the check, *ld.c* is simply executed as a no-op. If there is a conflict, the ld.c instruction will reload of the up-to-date data from memory. Therefore, the correctness of speculative register promotion is guaranteed.

Another case is a redundant load following a store operation. In this case, a *ld.a* instruction is added after the store instruction to secure an entry in ALAT (shown in Figure 1(b)). When there are multiple 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 1(c).

2.3. Transformations with control flows

For partially redundant loads, such as the second load in Figure 2(a), 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 at a dominating point to handle partial redundancy, as shown in Figure 2(b). 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 lifetime of a register. However, with a large register file as in Itanium, the register pressure is usually not a big problem.

When data is reused across an entire loop (see Figure 3(a)), 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 could be used here. We only need one check instruction, such as *chk.a*, to check both control and data speculation. The check instruction should keep the entry in ALAT (i.e. the *ld.c* flag should be set to not-clear) because each of the subsequent iterations needs to use the allocated ALAT entry. Figure 3(b) shows the code to speculatively promote a speculative loop invariant to a register



a. Original Code b. Speculative register promotion

Figure 2. An example of if statement

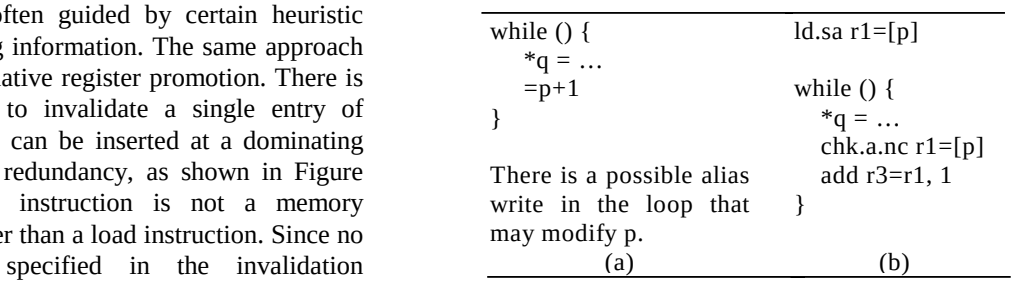


Figure 3. An example of loop

2.4. 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 [25]. Such cases may happen when the address part of the reference may have aliased writes. For instance, **p*, *a[j]* and ***q* 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 4(a) shows the source code of such an example. Figure 4(b) shows the transformation when only the address may be modified, and Figure 4(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.*

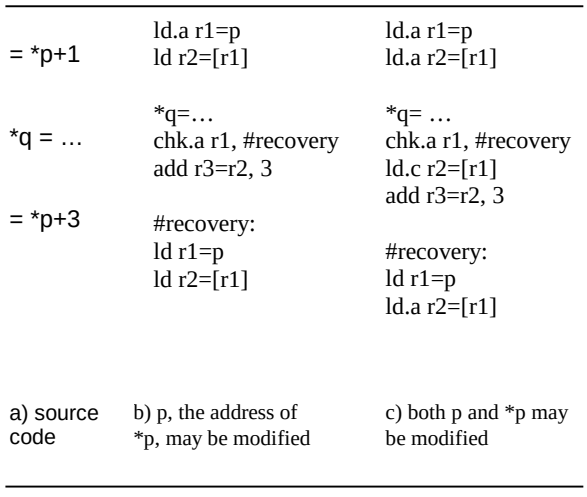


Figure 4. Example of cascade failure

2.5. 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* allocates an entry in ALAT.

The following data reads must be checked with *ld.c* or *chk.*a. The *ld.*c can be executed concurrently with the consumer instructions, and takes zero cycle if there is no collision. The *chk.a* may not be scheduled in the same bundle with its consumers However, its recovery scheme supports 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 reload the data from the memory and the load latency will be 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 an unconditional branch. Therefore, a mis-speculation, especially for address mis-speculation, could be expensive.

3. Algorithm for speculative register promotion

In this section, we discuss the algorithm used to generate code for speculative register promotion, based on the scheme described in the previous section. Our algorithm is designed based on partial redundancy elimination (PRE) so that control speculation can also be handled. Though the algorithm is described in the context of the ORC Compiler's infrastructure, it can be applied to other compilers. The ORC compiler adopts a powerful SSA form to model the indirect references [23, 33]. The PRE algorithm used in ORC is SSA based, called SSAPRE [29]. We will start from how to incorporate alias speculation in the SSA form for speculative register promotion.

3.1. Speculative SSA form

The original SSA form in the ORC compiler is designed to represent indirect reference more precisely. It is a location-factored representation enhanced by the use of virtual variables [23, 33]. The update and the use operations of indirect references are modeled by χ and μ operations.

Since the static pointer analysis tends to be conservative, we try to speculate the alias relationship in a program to perform more aggressive register promotion. In this paper, we focus on using the alias profiling feedback for alias speculation. Other speculation methods, such as using heuristic rules, can also applied in this framework. We developed a tool on the top of ORC to instrument the code, and then, to collect the target set of every memory load or store operation at runtime [7, 8].

To represent the feedback from alias profiling in SSA form, we introduce a new notion called speculative update and speculative use. A *speculative* flag is added to these operations to indicate that, according to the alias profile, these operations may not occur at runtime and can be speculatively ignored. The two new speculative operations are denoted χ_s and μ_s, respectively. Figure 5 gives an example of how to determine the speculative flags using

alias profiling feedback [7,8]. Those updates or uses related to the targets that do not appear in the alias profile are marked speculative.

*p = $b_2=\chi(b_1)$ $a_2=\chi_s(a_1)$ $v_2=\chi_s(v_1)$	$\mu(b_1)$ $\mu_s(a_1)$, $\mu_s(v_1)$ = *p
The two examples assume that the points-to set of p generated by compiler is {a, b}, the points-to set of p obtained from alias profiling is {b}. v is the virtual variable for *p. a_j stands for version j of the variable a.	

Figure 5. Determine the speculative flag according to alias profiling

3.2. Overview of register promotion based on PRE

In order to identify more candidates for register promotion in the context of control-flow structures, register promotion is performed based on partial redundancy elimination (PRE). There are two existing partial redundancy elimination schemes: one is bit-vector based [32] and the other is SSA form based [22, 29]. In this paper, we focus on the SSA form based PRE (SSAPRE) because it is adopted in the ORC compiler and our implementation is based on ORC.

In SSAPRE, each *expression* is processed in a bottom-up order in its syntax tree. For example, in the syntax tree **p, p is processed first, then *p, and finally followed by **p. When an *expression* is processed, the SSA form for the values of this *expression* (called *hypothetical temporaries* in SSAPRE) is constructed based on the variable SSA form [29]. Therefore, the occurrences of an *expression* with the same value can be identified. These *expressions* may be redundant. For partially redundant expressions, computations are added along the incoming path with control speculation so that the partial redundant computations become full redundant and can be eliminated. In [29], there are 6 steps to identify redundant expressions: 1) Phi-insertion, 2) Rename step, 3) DownSafety, 4) WillBeAvail, 5) Finalize and 6) CodeMotion.

The first two steps are aimed to identify the expressions that have the same value and are redundant. The Phi-insertion step marks every point at which the value of an expression may change, namely, the update points for the hypothetical temporaries. The version numbers for the hypothetical temporaries are assigned in the Rename step. The following step 3 and step 4 are intended to handle the partial redundancy in the control flow graph. The DownSafty step checks whether a hypothetical temporary may not be used later, and the WillBeAvail step checks whether the value of a hypothetical temporary is available from all the incoming paths. The Finalize step determines the placement of computation. The last step, the CodeMotion step transforms the code. More details can be found in [29].

Our speculative register promotion work directly affects the Rename step and the CodeMotion step. By ignoring some points-to targets and consequent updates, some occurrences of the hypothetical temporaries may speculatively have the same version number and more redundancy can be identified. In the CodeMotion step, we generate speculative load and check instructions. We do not modify the partial redundancy work because the data speculation is orthogonal to the control speculation. Details are discussed in the following sections.

3.3. Speculative rename step

In the Rename step, version numbers are assigned to the hypothetical temporaries. To assign version numbers, the Rename step keeps track of the current version of the expression and the variables contained in the expression by maintaining a rename stack for each of them while conducting a preorder traversal of the dominator tree of the program. The critical operation is to compare whether all the variables in the current occurrence of the expression have the same version number as those variables in the expression on the top of the rename stack.

There are two cases that may cause the version number to be changed: the merge of control flow or the update of a variable contained in the expression. With data speculation, there are two kinds of updates that may be ignored: 1) speculative updates χ_s, and 2) updates related to the speculative use μ_s. These updates are unlikely to change the value of the hypothetical temporaries according to the speculation. As a result, more occurrences of the expression are assigned with the same version number speculatively in our original algorithm. In order to generate correct code in the CodeMotion step, we attach a speculative flag to the version number if it is assigned speculatively.

Here we use the example in Figure 6(a) to show how the speculative Rename step works. In this example, there are two occurrences of the *expression a* which are represented by the temporary variable *h*. The alias analysis shows that there exists alias relation between the expression *p and *a*. The value of the variable *a* is updated which is represented by χ operation after the store of *p in the resulting SSA form. These two occurrences of *a* are assigned with different version numbers in the original Rename step. However, in our algorithm, if p

doesn't point to *a* in the alias profile or from some heuristic rules, the χ operation with *a* is marked with χ_s and this update can be ignored in the Rename step. In Figure 7 (b), the second occurrence of *a* is speculatively assigned with the same version number as the first one. In order to generate the check statement in the CodeMotion step, the second occurrence of *a* is annotated with a speculative flag.

The target set of *p generated by the compiler is {a, b} and v is the virtual variable for *p.
The target set of *p generated by the alias profiling is {b}.
h is the hypothetical temporary for the load of a.

… = a_1 [h_1]	… = a_1 [h_1]
*p_1 = …	*p_1 = …
$v_2 \leftarrow\chi$ (v_1)	$v_4 \leftarrow\chi$ (v_3)
$a_2 \leftarrow\chi$ (a_1)	$a_2 \leftarrow\chi_s$ (a_1)
$b_2 \leftarrow\chi$ (b_1)	$b_2 \leftarrow\chi$ (b_1)
… = a_2 [h_2]	… = a_2 [h_1<speculative>]
a) traditional Renaming	(b) speculative Renaming

Figure 6. Example of speculative renaming

3.4. Generate advanced load and check instructions

The CodeMotion step transforms the code according to the SSA form for the hypothetical temporaries built in the previous steps. In this step, when there are reuses, the corresponding hypothetical temporaries become real temporaries in order to hold the value for reuses. Other hypothetical temporaries should be discarded. Assignments to the real temporaries and the use of the real temporaries are generated in CodeMotion step.

With data speculation, this step is responsible for generating speculative load for the assignments to the real temporaries and generating the check statements for the uses of the real temporaries. The check statements are needed at places where the speculative occurrence is anticipated. At the same time, redundant checks should be removed as much as possible.

Figure 7 gives an example that shows the effect of the algorithm. In this example, the second occurrence of *a* is annotated with a speculative flag in the Rename step to indicate that the version number of the temporary variable *h* is speculatively identical to the version number of the first occurrence. In the original CodeMotion step, the temporary variable t is generated to model the hypothetical temporary variable *h*, the first occurrence of *a* is replaced with an assignment statement to variable *t*, and the second occurrence *a* is replaced with the use of variable t according to the result of the Finalize step. In addition, in our enhanced CodeMotion step, we insert an assignment statement to *t* after the store of *p. This statement is called check statement in this paper. The expression *a* on the right hand side of check statement is marked with ld.c flag. Since the value of the first occurrence of *a* can reach the second occurrence, the expression *a* at the first occurrence is marked with a ld.a flag. The ld.a and ld.c flags are used to guide later code generation.

… = a_1 [h_1] *p_1 = … $v_4 \leftarrow\chi$ (v_3) $a_2 \leftarrow\chi_s$ (a_1) $b_4 \leftarrow\chi$ (b_3) … = a_2 [h_1<speculative>] (a)Before Code Motion	t_1 = a_1 (ld.a flag) … = t_1 *p_1 = … $v_4 \leftarrow\chi$ (v_3) $a_2 \leftarrow\chi_s$ (a_1) $b_4 \leftarrow\chi$ (b_3) t_4 = a_2 (ld.c flag) … = t_4 (b) Final Output

Figure 7. Example of speculative code generation

3.5. Recovery code generation

In this phase, all the statements marked with speculative load and check flags are transformed into the corresponding assembly instructions. The recoveries codes are generated to ensure the correctness of the original program should mis-speculation occur.

We used the recovery code generation approach introduced in [21]. It can generate recovery code for speculation during list scheduling as well as in other optimization phases.

4. Performance evaluation of the speculative register promotion using ALAT

We implemented the speculative register promotion based on PRE in the Intel's Open Research Compiler (ORC) of version 1.1. The benchmarks used are selected from SPEC CPU2000 benchmarks. Experiments were conducted on an HP-i2000 workstation with a 733 MHz Itanium processor with Linux 7.1 operation system.

The base line used for comparison in our experiments is the code generated by ORC with –O3 option. In the base line version, ORC performs a sequence of pointer analyses, such as equivalence class based alias analysis [24], flow sensitive pointer analysis and even the unsafe type-based pointer analysis. Based on these alias analysis results, ORC performs a powerful register promotion based on partial redundancy elimination. The software approach to check possible alias by compare instructions at runtime [30] is also applied. Therefore, the further

improvement in register allocation can be regarded as the contribution of our speculative register promotion.

In the experiment, the alias profiling information is collected with the *train* input set, and then is fed back to compiler to perform speculative register promotion. The generated code is executed with the *ref* input. The performance of the generated code is measured with the pfmon [28] tool.

The performance result of ten benchmarks is reported in Figure 8. We measure improvement using several metrics: the total CPU cycles, the data access cycles and the number of retired load instructions. As shown in Figure 8, the total number of CPU cycles are reduced by 1% to 7%. The major contribution to the reduction of CPU cycles comes from the reduction of data access cycles, and the reduction of data access cycles comes from reduced load instructions. The number of retired load operations are reduced by more than 5% for half of the benchmarks. The reduced load instructions are likely to be cache hits. On Itanium, the latency of a L1 data cache hit is two cycles. If those eliminated loads could be scheduled to fully hide the two-cycle latency, then the performance improvement would be minor. This explains why more than 5% of load instruction reduction only translates into 1% to 7% of CPU cycle reduction. In some cases, the eliminated loads reduce data cache misses, and contribute more to the performance gain. As shown in Figure 8, the performance gain of floating point benchmarks (ammp, art, equake) is more significant than integer benchmarks. This is because the latency of a floating point load on Itanium is 9 cycles. Converting 9 cycle loads to 0 cycle checks can contribute significantly to performance.

To further evaluate the impact on reduced loads for speculative register promotion, we study the relative percentage of indirect load and direct load among the reduced loads for each benchmark. In Figure 9, we observe that the indirect loads account for the majority of the reduced load for *ammp*, *gzip*, *mcf*, and *parser*.

We also report the percentage of load checks that failed during runtime over the total loads retired, which indicates the amount of data speculation opportunities having been exploited in each program. We also reported the percentage of check loads that failed during runtime, and this metric is called the *mis-speculation ratio*. A high mis-speculation ratio can decrease the benefit of speculative optimization or even degrade performance. In Figure 10, we observe that the mis-speculation ratio is generally very small. For *gzip*, although the mis-speculation ratio is at 5%, the total number of check instructions is nearly negligible compared to the total number of load instructions. Therefore, there is little performance impact from the high mis-speculation ratio.

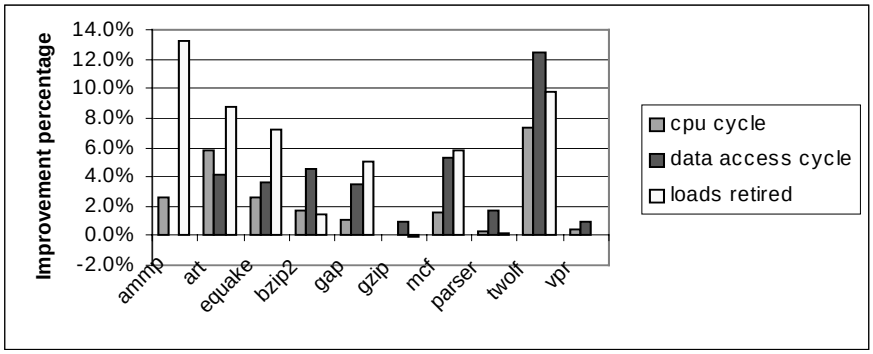


Figure 8. Performance of speculative register promotion

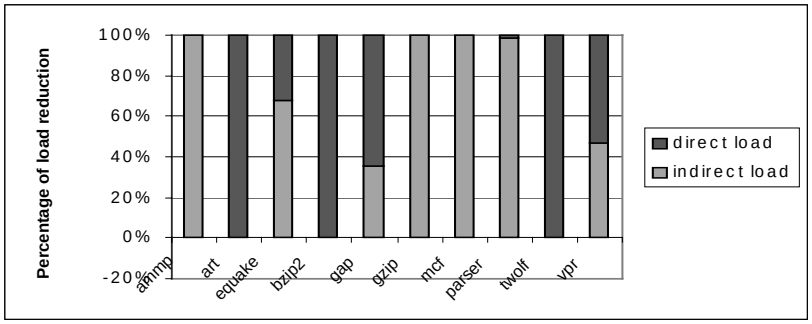


Figure 9. Percentage of different types of load among total reduced load

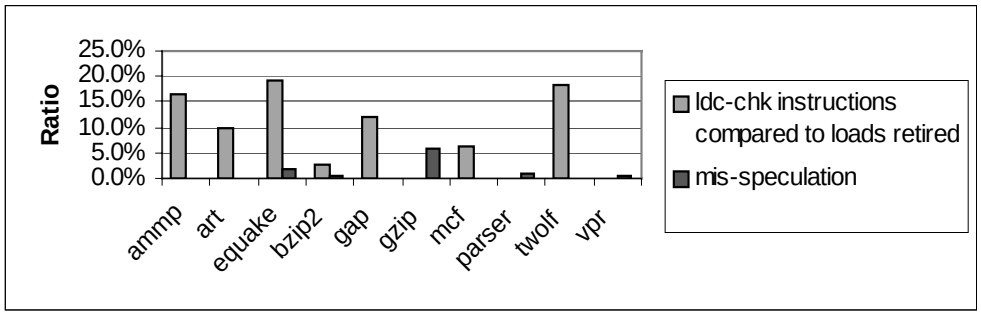


Figure 10. The mis-speculation in speculative register promotion

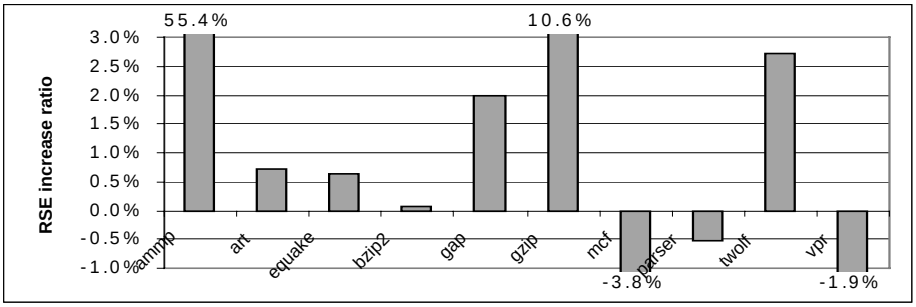


Figure 11. RSE memory cycles increase

Another concern for speculative register promotion is that it might increase the register pressure, thus cause excessive register spilling. On Itanium, register spilling has not been a major issue because of the use of a flexible register stack. Increased use of register will result in a larger number of registers allocated at the procedure prologue. Register stack overflow is handled by the RSE (Register Stack Engine). In Figure 11, we can see that the RSE cycles reported are barely changed, which indicates essentially very little increased register save/restore cost. For *ammp and gzip*, although the increase of RSE cycles is up by 55.4% and 10.6%, respectively, the total RSE cycles are only 0.001% of the total execution cycles. Hence, increased RSE cycles can be ignored.

The implementation of the speculative register promotion is not a trivial task. It has a large amount of inter-dependence with other optimizer components such as the instruction scheduler, the loop optimization phase, the software pipeliner, and the regular register allocation. In the current version, such interaction among optimizations has not been tuned, and the register promotion is limited to expressions that will not cause cascaded failure. The potential of speculative register promotion would be higher than what we have achieved.

5. Related work and discussions

Several hardware designs have been proposed 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. C-reg [13] is another design that simplifies the register forwarding. The Store-Load Address Table (SLAT) [14] supports speculative register promotion in which a separate table records the addresses of data that have been allocated to registers. These schemes maintain the coherence of values in registers when memory aliases do exist. The ALAT scheme uses a recovery mechanism to restore the correct value when the check instructions failed. As a result, ALAT requires fewer entries than the register file, and is not part of the architecture states. Partial addresses are used in ALAT to be space and time efficient while other schemes must use full address. SLAT can only promote scalar variables while ALAT can be used to promote both scalar and indirect references. One of the drawbacks of the ALAT scheme is that only load operations can be reduced.

The alias problem in speculative register promotion can also be solved by pure software approach [30]. 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 by hardware. The software approach has to generate an explicit compare instruction for every possible alias reference after every aliased store operation. It is enabled at O3 level in ORC and our results include this optimization.

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.

The potential of using data speculation in register promotion has been suggested by [22]. We present a compiler framework to support speculative register promotion. Our design and implementation of alias speculation is based on the SSA form as proposed in [23]. This SSA form models indirect references more precisely than the assignment-factor representation and more efficient than other location-factor representations. We introduce a new concept, the alias speculation, into the SSA form. Speculative register promotion is one optimization that can make effective use of alias speculation.

Our register promotion algorithm is based on the partial redundancy elimination algorithm [6, 29]. The conventional PRE algorithm focuses on how to use control speculation to eliminate partially redundant operations, regarding to the control flow paths. Now we can also identify speculative redundancies with the alias speculation support.

6. Summary

Register promotion 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), as 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 compiler algorism, based-on PRE, for speculative register promotion is presented. This algorithm is able to perform both control and data speculation in register promotion.

The Intel ORC compiler is used to evaluate the performance of speculative register promotion using ALAT. The PRE part in ORC is modified to perform speculative register promotion. The experiments show that the proposed scheme is quite effective on a set of selected benchmark programs. The integration of this scheme with other optimization phases in the ORC compiler is currently under development.

Acknowledgements

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

The authors wish to thank Roy Ju, Sun Chan (Intel Corporation), Raymond Lo, Shin-Ming Liu (Hewlett-Packard) and Peiyi Tang (University of Arkansas at Little Rock) for their valuable suggestions and comments, and the conference reviewers, whose comments significantly improved this paper.

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] F. Chow and J.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] R. Lo, F. Chow, R. Kennedy, S. Liu, and P. 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] T. Chen, J. Lin, W.C. Hsu and P.C. Yew, On the Impact of Naming Methods for Heap-Oriented Pointers in C Programs, In Proceedings of the International Symposium on Parallel Architectures, Algorithms, and Networks, 2002

[8] T. Chen, J. Lin, W.C. Hsu and P.C. Yew, "The Empirical Study of the Granularity of Pointer Analysis in C programs. In Proceeding of the 15th Workshop on Languages and Compilers for Parallel Computing, 2002.

[9] R.P. Wilson and M.S. Lam. Efficient context-sensitive pointer analysis for C program. In Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation, pages1-12, La Jolla, California, Jun 18-21, 1995.

[10] Michael Hind. Pointer analysis: Haven't we solved this problem yet? In 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'01), Snowbird, UT, June 2001.

[11] R. P. Wilson and M. 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] M. Postiff, D. Greene, Greene and T. 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] R. D.-C. Ju, S. Chan, and C. 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] P. Dahl and M. O'Keefe. Reducing Memory Traffic with CRegs. In Proceeding of the 27th International Symposium on 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] E.Morel and C. Renvoise. Global optimization by suppression partial redundancies. Comm. ACM 22(2): 96-103, February 1979.

[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. 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. 2000.

[23] F. Chow, R. Lo, S. Liu, S. Chan, and M. 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] B. Steensgaard. Points-to analysis in almost linear time. In Proceedings 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] R. Ghiya, D. Lavery and D. Sehr. On the Importance of Points-To Analysis and Other Memory Disambiguation methods For C programs. In Proceedings of the ACM SIGPLAN'01 Conference on Programming Language Design and Implementation, page 47-58, June 2001.

[28] pfmon: ftp://ftp.hpl.hp.com/pub/linux-ia64/pfmon-1.0-1.ia64.rpm

[29] R.Kennedy, S. Chan, S. Liu, R. Lo, P. Tu, Partial Redundancy Elimination in SSA Form. ACM Trans. On Programming Languages and systems, 1999.

[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 Proceeding of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems, pages 183-193, Oct. 1994.

[32] J. Knoop, O. Ruthing, and B. Steffen. Lazy code motion. In Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation, San Francisco, California, June 1992.

[33] R. Ju, S. Chan, F. Chow, X. Feng, Open Research Compiler (ORC): Beyond Version 1.0, tutorial presented in PACT 2002.