A General Compiler Framework for Speculative Optimizations Using Data Speculative Code Motion

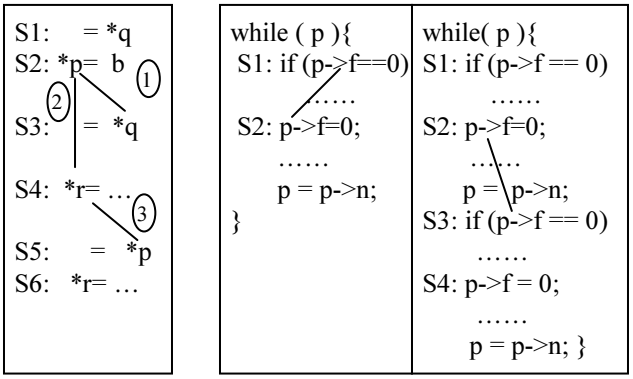
Xiaoru Dai, Antonia Zhai, Wei-Chung Hsu, Pen-Chung Yew
Department of Computer Science and Engineering
University of Minnesota
Minneapolis, MN 55455
{dai, zhai, hsu, yew}@cs.umn.edu

Abstract

Data speculative optimization refers to code transformations that allow load and store instructions to be moved across potentially dependent memory operations. Existing research work on data speculative optimizations has mainly focused on individual code transformation. The required speculative analysis that identifies data speculative optimization opportunities and the required recovery code generation that guarantees the correctness of their execution are handled separately for each optimization. This paper proposes a new compiler framework to facilitate the design and implementation of general data speculative optimizations such as dead store elimination, redundancy elimination, copy propagation, and code scheduling. This framework allows different data speculative optimizations to share the followings: (i) a speculative analysis mechanism to identify data speculative optimization opportunities by ignoring low probability data dependences from optimizations, and (ii) a recovery code generation mechanism to guarantee the correctness of the data speculative optimizations. The proposed recovery code generation is based on Data Speculative Code Motion (DSCM) that uses code motion to facilitate a desired transformation. Based on the position of the moved instruction, recovery code can be generated accordingly. The proposed framework greatly simplifies the task of incorporating data speculation into non-speculative optimizations by sharing the recovery code generation and the speculative analysis. We have implemented the proposed framework in the ORC 2.1 compiler and demonstrated its effectiveness on SPEC2000 benchmark programs.

1. Introduction

Imprecise data dependence information may decrease the effectiveness of compiler optimizations. However, obtaining precise data dependence analysis is both difficult and expensive for languages such as C in which dynamic and pointer-based data structures are frequently used. When the data dependence analysis is unable to show that there is definitely *no* data dependence between two memory references, the compiler must assume that there *is* a data dependence between them. It is quite often that such an assumption is overly conservative. The examples in Figure 1 illustrate how such conservative data dependences may affect compiler optimizations.



a) Example 1 **b) Example 2**

Figure 1. Examples of compiler optimizations disabled by possible data dependences.

In Figure 1, lines represent possible data dependences between memory references. For the example in Figure 1 (a), the possible true dependence between *p and *q (line 1) prevents possible redundancy elimination of *q in S3. The possible output dependence between *p and *r (line 2) inhibits possible copy propagation of *p in S5. The possible true dependence between *p and *r (line 3) disallows possible dead store elimination in S4. In this example, three compiler optimizations (redundancy elimination, copy propagation, dead store elimination) are inhibited by possible data dependences. Without these

Proceedings of the International Symposium on Code Generation and Optimization (CGO'05)
0-7695-2298-X/05 $ 20.00 IEEE

possible dependences, those optimizations could have been performed by the compiler.

In the left column of Figure 1 (b), another motivation example with a C *while* statement is shown. In this example, the *load* of p->f in S1 may have possible data dependence with the *store* of p->f in S2. After the loop is unrolled, the result code is shown in the right column of Figure 1 (b). S1 and S2 are from the first iteration, S3 and S4 are from the second iteration. The *load* of p->f in S3 cannot be scheduled ahead of the *store* in S2 because of the possible data dependence. If this data dependence rarely happens at runtime, it may be profitable to schedule the *load* in S3 before the *store* in S2 to hide the *load* latency. If the data dependence indeed happens, a recovery code needs to be executed to guarantee the correct results.

Getting precise data dependence information is difficult because it is hard for a compiler to know what memory locations a memory reference may access at run time. It is even more difficult when pointers are involved in the program. Therefore, using *data speculation* and *runtime verification* to overcome possible data dependences (with low probabilities) has been proposed recently in [11-16]. Here, *data speculation* refers to the execution of instructions which may potentially violate possible memory dependences albeit infrequently.

Compiler optimizations are normally divided into two phases: the analysis phase and the code transformation phase. The analysis phase identifies optimization opportunities based on the internal representation (IR) and data dependence information. The code transformation phase modifies IR to generate improved code. To support data speculation, we need a recovery mechanism using either hardware or software support to guarantee the correctness of their speculative optimizations.

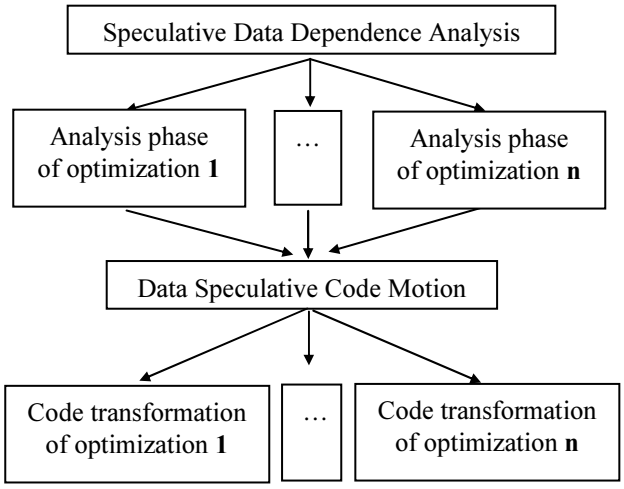


Figure 2. Structure of our proposed data speculative optimizations

The work in [12] and [15] uses data speculation in code scheduling to generate more efficient code sequence. In [13][14][22], data speculation is used to enable speculative register allocation. They are all examples of specific speculative code optimizations.

In [11], Ju et al. proposed a unified compiler framework for control and data speculation in a code scheduler. There are three main tasks in their speculative code scheduler: marking speculative dependence edges, selecting speculative instructions as scheduling candidates, and *check* insertion and DAG update. These three tasks are integrated with the rest of the instruction scheduling phase.

In [16], a framework that augments SSA form to incorporate data speculative information (obtained either from alias profiling or compiler heuristic rules) is proposed. Speculative partial redundancy elimination based on the SSAPRE [5] is presented to exemplify the use of such a framework.

In both [11][16], the data speculative information is *explicitly annotated* either through speculative dependence edges in dependence graph [11] or speculative weak updates in SSA form (i.e. χ and μ operators in [16]). All optimizations that try to incorporate data speculation thus must be modified and made aware of such explicitly annotated data speculative information.

In [11], the construction of dependence graph, selection of scheduling candidates and DAG update are all modified to handle the speculative dependence edges. Recovery code generation is decoupled from the scheduling phase, and works well only for code scheduling. It may not handle other optimizations directly. For example, the identification of speculative chains in their recovery code generation will not be applicable for eliminating instructions due to speculative redundancy. In [16], the construction of SSA form, the Φ-insertion step, the rename step and the code motion step in SSAPRE all need be modified to identify speculative optimization opportunities and to generate recovery code. In [11][16], the accommodation of data speculative information in optimizations and the recovery code generation have to be tailored to each specific compiler optimization. They can't be shared among optimizations. Such existing frameworks are difficult to adopt, to extend, and to maintain.

In our framework, as shown in Figure 2, the data speculative information is integrated into a *shared Speculative Data Dependence Analysis* (SDDA) phase by ignoring *low probability* data dependences from the optimizations. Hence, more optimization opportunities could be exposed for existing optimizations without requiring any modification to accommodate such information as in [11] and [16]. When an optimization opportunity is identified in the analysis phase of an optimization, a *shared* mechanism is provided for

recovery code generation if it is data speculative. The proposed recovery code generation is based on *Data Speculative Code Motion* (DSCM) which uses a code motion model to determine whether a transformation is data speculative, and to generate necessary recovery code.

Our framework has two advantages. First, SDDA and DSCM are *shared* by all optimizations. Second, the existing non-speculative optimizations need *no* modifications. There is no need to make customized changes in each optimization to accommodate speculative information and to generate recovery code as in [11] and [16]. To show the effectiveness of such an approach, we have successfully applied our framework to four optimizations: redundancy elimination, dead store elimination, copy propagation, and code scheduling.

1.1 Contributions

This paper focuses on and is made novel by proposing a data speculative code motion framework that transforms existing data dependence based optimizations into data speculative optimizations. Prior work on data speculation all targets specific optimizations, such as code scheduling [11, 12, 15] or register allocation [13, 14, 22]. The only prior work that attempts to integrate different compiler optimizations into a common framework is [16]. However, it requires two major modifications whenever a new optimization is integrated: (i) an optimization pass that accommodates data speculation information, and (ii) a recovery code generation mechanism tailored to that specific optimization. The framework proposed in this paper is the first attempt to allow the easy integration of different data speculative optimizations by sharing the same speculative data dependence analysis and recovery code generation while keeping the corresponding non-speculative optimizations intact by using data speculative code motion.

The rest of the paper is organized as follows. Section 2 describes the heuristic rule in SDDA. Section 3 discusses code motion required by each optimization in details. Section 4 explains DSCM and its recovery code generation. Section 5 provides the implementation of our framework. Section 6 discusses the performance of the framework using SPEC CPU2000 benchmarks, and section 7 presents our conclusions.

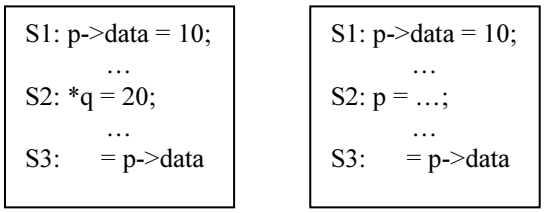
2. Speculative Data Dependence Analysis

Traditional data dependence analyses must conservatively assume that there is a data dependence between two memory references unless it can be proven otherwise, while our speculative data dependence analysis (SDDA) does exactly the opposite and takes a very aggressive approach. SDDA assumes that there is no data dependence between two memory references unless we could prove that it is *very likely*, or *most definitely*, that those two memory references *will* access the same memory locations. Any data dependence with a low probability will be assumed as *no* data dependence in the speculative optimizations. As it turns out, the probability distribution of most data dependences are very bimodal, i.e. it is either very likely, or not likely at all [20]. Using this approach, more optimization opportunities can be exposed for possible speculation. However, since SDDA cannot guarantee the correctness of the execution, we still need the results from the traditional data dependence analysis to guide the recovery code generation for all speculative optimizations based on SDDA.

We use *access paths* [1] to represent memory references. An *access path* (AP) of a memory reference is a non-empty string that consists of the variable name, the field name of a structure and the de-reference to reach the memory location of the memory reference. We use rules similar to those in [1] to generate access paths for memory references.

A heuristic rule is used to identify *highly likely* dependent memory references. The simple heuristic rule used is as follow: if two memory references have the same access path, and if the variables involved on the access path are not explicitly changed between the two references, then the two references are considered *dependent*. Otherwise, they are considered *independent*. Data dependence profile [20] could also be used to provide such information.

In Figure 3 (a), p->data in S1 and in S3 are considered *dependent* since they have the same access path and the variable p is not explicitly changed between S1 and S3. p->data and *q are considered *independent* since their access paths are different. In Figure 3 (b), p->data in S1 and in S3 has the same access path. However, they are considered as *independent* because p is explicitly changed between S1 and S3.



a) Dependent memory references **b) Non-dependent memory references**

Figure 3. Heuristic rule in SDDA

In SDDA, access path information is collected for every memory reference in a procedure. After that, data dependence relations between any two memory references are computed based on the above heuristic rule.

3. Code Motion to Enable Data Speculative Optimizations

When an optimization uses the results of SDDA, it may violate some unlikely, but possible, data dependences at runtime. Therefore, we must generate recovery code to guarantee the correctness of the data speculative optimizations. In this section, we analyze four data speculative optimizations to show that the correctness of data speculative optimizations can be guaranteed by a special code motion. We explain this in details for *data speculative dead store elimination* in section 3.1. The other three data speculative optimizations (*redundancy elimination*, *copy propagation* and *code scheduling*) can be explained in a similar way. We provide a brief explanation for them in section 3.2, 3.3 and 3.4.

3.1 Code Motion to Enable Data Speculative Dead Store Elimination

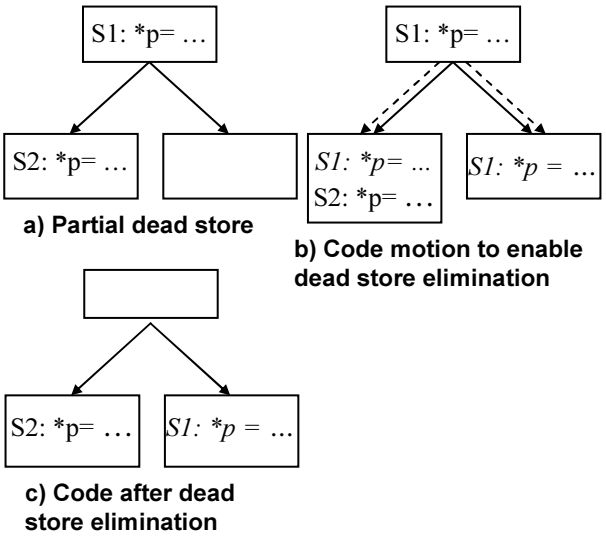


Figure 4. Code motion to enable data speculative dead store elimination

Dead store elimination [17][6] eliminates unnecessary *stores* of which the results are not used. For the example in Figure 4 (a), the analysis phase of the *dead store elimination* identifies that the *store* in S1 is *dead* on the left path if there are no highly likely aliased *loads* or *stores* between S1 and S2. But the *store* cannot be eliminated directly because there may be possible aliased *loads* or *stores* along the left path (Note: the optimization uses the results of SDDA). We consider the *store* as a speculative optimization opportunity. A special code motion that moves the *store* in S1 to both subsequent blocks can convert the speculative optimization opportunity to non-speculative optimization opportunity that guarantees the correctness of the speculative optimization as shown in Figure 4 (b). The dotted lines show the direction of the code motion. The question here is whether the *store* in S1 can be moved to the target blocks as described above. This code motion may be illegal when possible data dependences exist along the code motion path. However, we could generate recovery code for every potential violation of memory dependences along the code motion path to guarantee the correctness of the code motion. The details of the code motion and the required recovery code generation are explained in section 4. After the code motion, the code transformation of *dead store elimination* can remove the *dead store* along the left path. The resulting code after *dead store elimination* is shown in Figure 4 (c).

3.2 Code Motion to Enable Data Speculative Redundancy Elimination

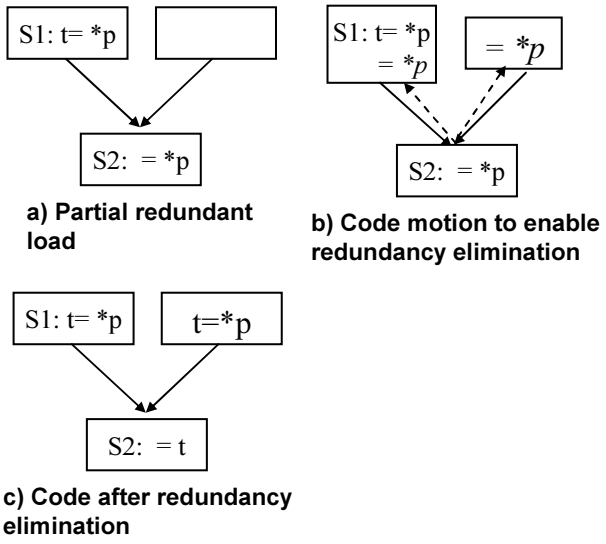


Figure 5. Code motion to enable data speculative redundancy elimination

Redundancy elimination [18][5] eliminates *loads* or *computations* of which the results are already available. For the example in Figure 5 (a), the *load* of *p in S2 is *redundant* when the execution follows the left path. After the analysis phase of the *redundancy elimination* identifies the *load* in S2 as *redundant*, our special code motion will move the *load* next to S1 as shown in Figure 5 (b). The *load* can be safely eliminated after the code motion. The code after *redundancy elimination* is shown in Figure 5 (c). In our implementation, the redundant load is replaced with a move instruction that copies the destination of the non-redundant load to the destination of

the redundant load. Thus the destination of the load in the recovery code (shown in section 4) does not need any modification which keeps the code motion independent of optimizations.

3.3 Code Motion to Enable Data Speculative Copy Propagation

Copy propagation replaces a *load* by a register access if a *store* to the same memory location can be found before the *load*. For example, in Figure 6 (a), the *load* of *p in S2 can obtain the result directly from t of the *store* instruction in S1. The analysis phase of *copy propagation* identifies that the *load* of *p in S2 can use the result of S1, our special code motion will move the *store* in S1 downward as shown in Figure 6 (b). The code after the *copy propagation* is shown in Figure 6 (c).

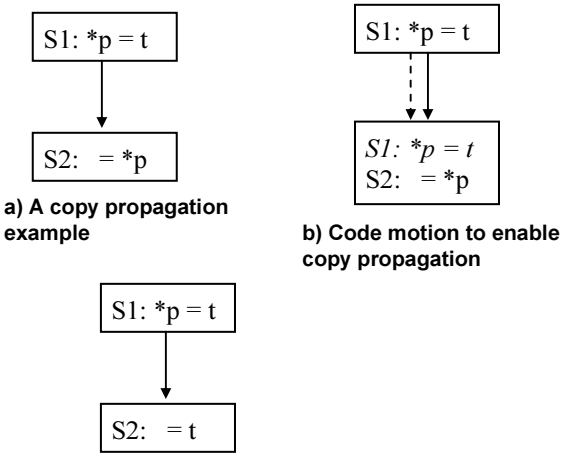


Figure 6. Code motion to enable data speculative copy propagation

3.4 Code Motion to Enable Data Speculative Code Scheduling

The code transformation of *code scheduling* is similar to *upward code motion*. Potential data dependences often limit the effectiveness of *code scheduling*. With SDDA, such constraints are relaxed since low probability dependences are ignored. The scheduler selects the best candidate to issue in each cycle. If any of the code motion violates potential data dependences, the verification and recovery code will be generated to ensure correct execution.

4. Six Cases of Data Speculative Code Motion

In section 3, we showed that the correctness of data speculative optimizations can be guaranteed by a special code motion. If the involved code motion does not violate any data dependences, no additional work is needed. Otherwise, recovery code must be generated to enforce the correctness of the code motion. We call such special code motion as *Data Speculative Code Motion* (DSCM).

Here, we limit the data dependences to only those caused by *memory references* since data dependences caused by *registers* will definitely happen at runtime, and shall not be violated. According to the *type* of the moved instruction (memory *load* or *store*), the *direction* of the motion (upward or downward) and the *type* of instruction being crossed by the code motion (another memory *load* or *store*), we classify DSCM into six cases listed in Figure 7. In each case, the dotted line represents the *direction* of the code motion and the solid line represents *control flow*. In each case, the two memory references are assumed to have possible data dependence. The summary of the six cases is given in Table 1. In this paper, ***ld*** represents a memory *load* and ***st*** represents a memory *store.*

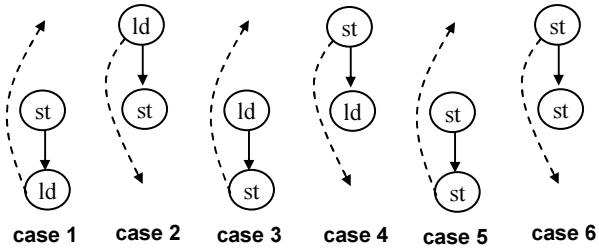


Figure 7. Six cases of DSCM

Although case 1 and 4, case 2 and 3 as well as case 5 and 6 are pair-wise similar in terms of code motion, they are different from the perspective of recovery code generation. We only generate recovery code for the moved instruction. The crossed instruction is not changed. Moving a *load* upward or downward across a potentially dependent *load* may violate input dependence [10]. Normally, violation of input dependence does not affect the correctness of the program. Hence, it is not taken into account in our framework.

Table 1. Summary of the six cases of DSCM

case #	moved inst	crossed inst	direction	dependence may be violated
1	ld	st	upward	true dependence
2	ld	st	downward	anti-dependence
3	st	ld	upward	anti-dependence
4	st	ld	downward	true dependence
5	st	st	upward	output dependence
6	st	st	downward	output dependence

4.1 Recovery Code Generation

Since DSCM may violate possible data dependences, verification code must be inserted to check if the DSCM is legal at runtime. If no dependence violation is detected at runtime, the DSCM is allowed. When the DSCM is not allowed, the associated recovery code must be executed. Comparing the memory addresses of the moved memory reference and the crossed memory reference can tell whether a data dependence is violated or not. What instructions should be in the recovery code and when should the recovery code be executed are dependent on each case of DSCM. Normally, the recovery code should include the speculatively moved instruction. If DSCM moves an instruction upward, we can execute the recovery code at the original place of the moved instruction. If DSCM moves an instruction downward, we may execute recovery code as soon as we detect a check failure in the verification code. There are four components in recovery code generation for the DSCM.

- Flag initialization: set a flag to indicate that DSCM is allowed.
- Verification: by comparing the addresses of the moved memory reference and the crossed memory reference, we could check if DSCM is allowed or not. If not, the flag is cleared (the flag is set initially).
- Insertion of check instruction: a check instruction is generated to check the flag. If the flag is cleared by verification code, execute the recovery code.
- Generation of recovery code: initially, only a copy of the moved instruction is included.

In Figure 8, we show the code segments, which include flag initialization, verification instructions, check instructions and recovery code, for case 1, case 4 and case 6, as examples to illustrate how DSCM is supported.

Case 1:	Case 4:	Case 6:
Before DSCM: st [r1] = r2 ld r3 =[r4]	Before DSCM: st [r1] = r2 ld r3 = [r4]	Before DSCM: st [r1] = r2 st [r3] = r4
After DSCM: S1: flag =1 S2: ld r3 = [r4] S3: if overlap(r1, r4) flag =0 S4: st [r1]= r2 S5: if (flag == 0) S6: ld r3=[r4]	After DSCM: S1: flag =1 S2: if overlap(r1,r4) flag =0 S3: if (flag==0) S4: st[r1] = r2 S5: ld r3 = [r4] S6: if (flag == 1) S7: st [r1] = r2	After DSCM: S1: flag =1 S2: if overlap(r1, r3) flag =0 S3: st [r3] = r4 S4: if (flag == 1) S5: st [r1] = r2

Figure 8. Code generation support of DSCM case 1, 4 and 6

Case 1 moves a *ld* upward across a possible dependent *st*. In the result code, S1 sets the flag to 1 to indicate the code motion is allowed. S2 executes the *ld* that is now ahead of the *st*. S3 verifies if DSCM is allowed or not. If the address (r4) of the *ld* and the address (r1) of the *st* overlap, the code motion violates the true data dependence, and the flag is cleared to indicate that the DSCM is not allowed. Detection of the overlap can be supported by hardware or software. S4 executes the *st* which is not changed. S5 is a check instruction to check the flag. If the flag is 0, indicating an illegal DSCM, then the recovery code S6 is executed. In this example, the recovery code is the original *ld*.

Case 4 is similar to case 1. The difference is that, after the verification instruction (S2), the check instruction (S3) is executed immediately to decide if recovery code (S4) should be executed or not because, if the DSCM is illegal, the following *ld* needs the result of the *st*. The check instruction (S3) can be eliminated since flag==0 is equivalent to detect the overlap between r1 and r4. In case 4, the moved instruction (S7) is guarded by a condition (S6). Whether we should execute the moved instruction or not depends on the legality of DSCM. If the DSCM is allowed, we should execute the moved instruction.

In case 6, we do not need the check instruction and the recovery code. If the DSCM is not allowed, we know that the crossed *st* overwrites the moved *st*. The moved *st* (S5) is guarded by a condition (S4) which is the same as in case 4.

5. Implementation of the Framework

We have implemented the proposed data speculative optimization framework in Intel's Open Research Compiler (ORC) [2] for the IA64 platforms [3] [8]. In our framework, first it conducts SDDA. Based on the results of SDDA, analysis phase of an optimization identifies speculative optimization opportunities. After that, information about what instruction should be moved and where the instruction should be moved to is given to the DSCM support routines. Based on this information, the DSCM support routines decide which one of the six cases it belongs to and generate recovery code accordingly. After that, the moved instructions may be eliminated during the code transformation of the optimization. In our framework, the SDDA and DSCM are shared by all optimizations. Currently, we include optimizations (redundancy elimination, copy propagation, dead store elimination) that are in the global optimization phase WOPT and the code scheduling in the code generation phase CG of ORC in our framework.

DSCM support routines use data dependence information provided by a conservative data dependence analysis (CDDA). DSCM generates recovery code when

the code motion crosses a data dependence indicated in CDDA but ignored in SDDA. In our implementation, the CDDA is based on ORC's alias analysis. The alias analysis in ORC has two parts. First part is a points-to analysis which is based on a non-standard type system [21]. Second part is a rule based alias analysis. A *base rule* assumes that two memory references are *not* aliased if their bases are different. An *offset rule* assumes that two memory references are *not* aliased if their bases are the same, and their memory access ranges defined by the offset and the size of the fields do not overlap. A *type rule* assumes that two memory references are *not* aliased if their types are different. For example, memory reference to type *float* cannot be aliased with a reference to type *integer*. The above three rules and some other rules are used in the ORC's rule based alias analysis. Based on the results of the alias analysis, CDDA assumes that there is a possible data dependence between two memory references if they are aliased.

The performance of data speculation framework is highly dependent on the target machine architecture. In our case, we take advantage of the two architectural features: *data speculative loads* and *predication*, available on the IA64 platforms to implement our DSCM support. In our implementation, the DSCM support routines can generate recovery code for case 1, 4 and 6. Case 2, 3 and 5 can be implemented in the same way. However, they are not needed in the four speculative optimizations presented here.

5.1 Case 1 of DSCM

Code before case 1 DSCM: st [r1] = … ld r2 = [r3] r4 = r2 + 1 Code after case 1 DSCM: ld.a r2 = [r3] st [r1] = … chk.a r2, recovery r4 = r2 + 1 Recovery: ld r2 = [r3]	Code after r4=r2+1 is moved across the chk.a: ld.a r2 = [r3] r4 = r2 + 1 st [r1] = … chk.a, recovery Recovery: ld r2 = [r3] r4 = r2 +1
a) A load is moved	**b) A computation inst is moved**

Figure 9. Recovery code for case 1

IA64 architecture includes instructions *ld.a* (*advanced load*) and *chk.a* (*advanced load check*) [3]. *ld.a* executes a *load* operation and it also stores the memory address of the *load* into the Advanced Load Address Table (ALAT). A *store* instruction will invalidate any entry in ALAT that has a memory address overlapping with the store address. The overlap detection is supported by hardware. *chk.a* checks the ALAT entry set by its corresponding *ld.a*. It will invoke the recovery code if the entry becomes invalid by an intervening *store*. An example is shown in Figure 9 (a). In this example, the moved *load* is changed to *ld.a*. At its original location, a *chk.a* instruction is generated. In this example, the recovery code is simply a copy of the moved *load* instruction, i.e. it needs to reload the data. In case 1 of DSCM, when a *computation* or a *load* instruction that directly or indirectly depends on a *chk.a* instruction that needs to be moved across the *chk.a* instruction, such an instruction needs to be added to the recovery code of the *chk.a* instruction. One such an example is shown in Figure 9 (b).

5.2 Case 4 and Case 6 of DSCM

Code before case 4 DSCM: st [r1] = r2 ld r3 = [r4] Code after case 4 DSCM: pr1, pr2 = cmp.ne r1, r4 [pr2] : st [r1] = r2 ld r3 = [r4] pr1: st [r1] = r2	Code before case 6 DSCM: st [r1] = r2 st [r3] = r4 Code after case 6 DSCM: pr1, pr2 = cmp.ne r1, r3 st [r3] = r4 [pr1]: st [r1] = r2
a) Recovery code for case 4	**b) Recovery code for case 6**

Figure 10. Recovery code for case 4 and 6

We use predicate registers in IA64 to hold the results of the verification instructions. There are 64 predicate registers in IA64. In IA64, a *compare* instruction can set two predicate registers at the same time. For example, pr1, pr2 = cmp.ne 2, 3 will set predicate register pr1 to 1 and pr2 to 0. Most instructions in IA64 can be guarded by predicate registers. If the predicate register of an instruction is set to 0, then this instruction is executed as a NOP. We use predication to generate more compact recovery code. For case 4, the resulting code is shown in Figure 10 (a). For case 6, the resulting code is shown in Figure 10 (b). In cases 4 and 6, we use software to detect the overlap between two memory references. In Figure 11, we only show the cases in which memory references are naturally aligned and have the same size. When memory references are un-aligned, using software to detect the overlap requires extra instructions and becomes more

complex. In the code generated by ORC, un-aligned memory references are very rare, so we don't allow data speculation on un-aligned memory references in case 4 and 6. When two memory references have different sizes such as word and byte, we will change the byte address to a word address by ignoring the two low bits of the byte address.

5.3 DSCM Crossing Multiple Dependent *st*s or *ld*s

In previous sections, we discuss DSCM only in the context of moving a single memory reference across another possible dependent memory reference. In real applications, the compiler may need to move a single memory reference across multiple *load*s and *store*s.

In case 1, if a *load* is moved upward across several possibly dependent *store*s, we only need to generate *one* check instruction at its initial location. If any of those *store*s accesses the same memory location as the *load*, the recovery code associated with the check instruction will reload the data. Hence, only one check instruction is needed for a *load* to cross multiple *store*s or *load*s in case 1.

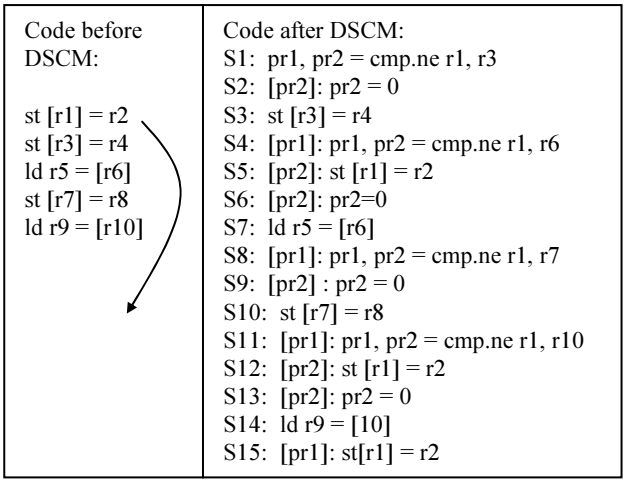


Figure 11. Recovery code for a st moved across multiple lds and sts

In case 4 and case 6, a *store* is moved downward across multiple *load*s and *store*s. The compiler needs to generate a check instruction for *every* crossed *load* and *store* with possible dependences. In Figure 11, we move the first *store* downward across two other *store*s and two *load*s. All verification instructions except the first one are guarded by the predicate register pr1, which will be set to 0 when the DSCM is not allowed. All check/recovery instructions are guarded by a predicate register pr2, which is set to 1 when the DSCM is not allowed. After recovery code is executed (in which the *store* is re-executed), pr2 is reset to 0. When pr1 and pr2 are both 0, all of the subsequent verification instructions and check instructions will be ignored to guarantee the correctness. We shouldn't execute the recovery code multiple times. In the example, S2, S6, S9, S13 are instructions which will set pr2 to 0 if pr2 is 1. After pr2 is changed from 1 to 0 (at this time, pr1 is already 0), all the following verification and check instructions are ignored.

In case 4 and case 6, we need to generate check instructions for every possible dependent memory references that are crossed by the code motion of the *st*. In order to avoid generating too many check instructions, we constrain the code motion in case 4 and case 6 not to cross procedure calls and loops.

6. Experimental Results

We evaluate the effectiveness of our framework on Itanium2 processor [8] using all the benchmarks from SPEC CINT2000 and all the C benchmarks from SPEC CFP2000. The runtime performance (in number of CPU cycles) for each benchmark is measured using *pfmon* [9]. All benchmarks are run with the *reference* input set. The base case for comparison is compiled with –O3 without data speculation by ORC. We compare optimizations with data speculation enabled by our framework to the base case in which they are not data speculative. In the following discussion, we list the percentage of runtime performance improvement, speculation fail rate, number of loads reduced, number of store reduced. We also show the average static number of dependent memory references crossed in each data speculative code motion opportunities in *mesa* and *crafty* are quite limited because most *store*s are to local variables. For all other benchmarks, the improvement ranges from 0.5% to 7%. The performance improvement of *speculative redundancy elimination* is marginal in integer benchmarks because the integer *load* latency is only 1 cycle when the *load* hits in L1 cache (a floating *load* has 6 cycles latency because the data comes from the L2 cache). Many redundant *load*s hit in the L1 cache, therefore, the saving of the redundant *load*s is relatively insignificant on Itanium 2. However, for future architectures with further increased clock rates, it may become harder to maintain a single clock hit latency for L1 Cache. The performance gain from redundant loads elimination will become more significant. The performance improvement of *data speculative code scheduling* on integer code is also not very impressive. This is because the linked list traversal often dominates memory references in the integer benchmarks. It is pretty difficult for the code scheduler to overlap memory references during the linked list traversal due to true data dependences. *Speculative dead store elimination* and *speculative copy propagation* do not yield much performance improvement in most benchmarks.

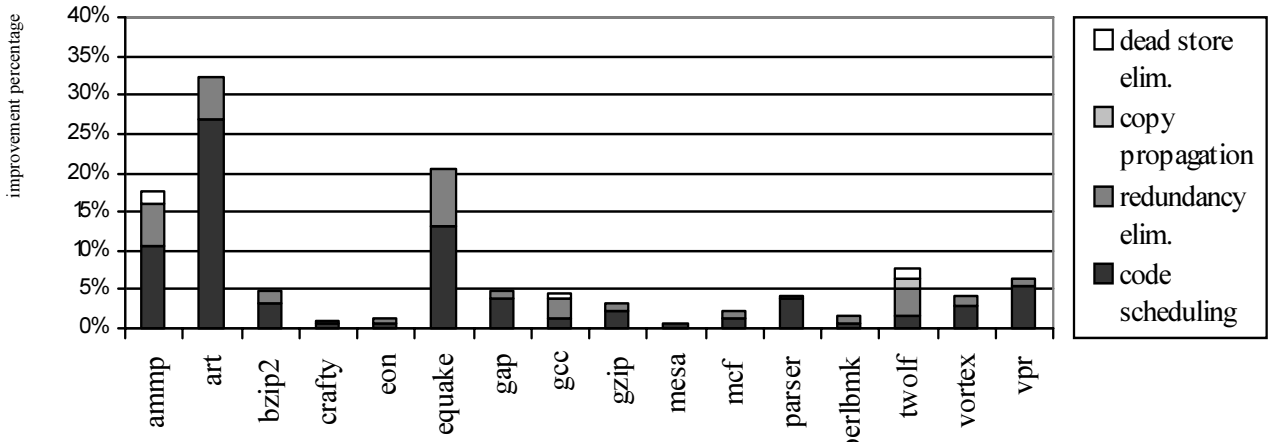


Figure 12. Performance impact of data speculative optimizations

Table 2. Static count of crossed dependent memory references

Benchmark	Dependent references	Benchmark	Dependent references
ammp	6.4	gzip	1.8
art	2.1	mcf	2.3
bzip2	1.5	mesa	1.8
crafty	1.6	parser	3.0
Eon	2.2	perlbmk	1.3
Equake	2.8	twolf	1.9
Gap	1.5	vortex	5.1
Gcc	1.2	vpr	2.7

In DSCM, when we move a memory reference across another data dependent memory reference, we need to verify the correctness of the code motion. In Table 2, we show the average static number of the dependent memory references crossed by each code motion. For most benchmarks, the number is smaller than 3.

Table 3 shows the *speculations fail rate* and the reduction of *load* and *store* instructions. We compute the *speculation fail rate* as the number failed *check* instructions divided by the number of *advanced loads*.

The penalty of executing recovery code could be very high because it may involve instruction cache misses for the recovery code and branch mis-predictions for the *check* instructions. However, since the speculation failure rate for most benchmarks is below 1%, we need not worry too much about the mis-speculation penalty. *Ammp* has a high failure rate (about 8%) because there are many *stores* between the *advanced load* and its corresponding *check* instruction that caused frequent false conflicts in ALAT(Note: ALAT only tracks *lower* address bits of an *advanced load*).

The reduction of *load* instructions indicates the potential of *speculative redundancy elimination*. From the measurements, floating-point benchmarks have a larger reduction in their *load* instructions. Normally, a larger *load* instruction reduction will result in a higher performance improvement.

Table 3. Speculation failure rate, reduction of loads and stores

Benchmark	Speculation failure rate	Number of *loads* reduced	Number of *stores* reduced
ammp	8.20%	17.20%	35.6%
art	0.02%	10.0%	<0.01%
bzip2	0.50%	1.30%	<0.01%
crafty	3.10%	0.12%	0.60%
eon	0.02%	-0.25%	<0.01%
equake	0.20%	31.6%	<0.01%
gap	0.06%	0.12%	<0.01%
gcc	1.14%	5.36%	2.60%
gzip	0.10%	0.60%	<0.01%
mcf	1.50%	4.60%	<0.01%
mesa	<0.01%	<0.01%	0.12%
parser	0.67%	0.20%	0.58%
perlbmk	0.03%	1.30%	0.15%
twolf	0.18%	5.80%	9.28%
tortex	0.17%	1.50%	<0.01%
vpr	0.02%	6.50%	0.31%

For the reduction of *store* instructions, only two benchmarks (*ammp*, *twol*f) have a large reduction of the *store* instructions. In the benchmarks we studied, *speculative dead store elimination* and *speculative copy propagation* do not yield much performance gain. However, in one time-consuming procedure of *ammp*, it identified many opportunities for *data speculative dead store elimination*. An example code is shown below.

```
S1: a1->VP += k*(*nodelist)[inode].q100;
S2: a1->dpx += k*(*nodelist)[inode].sqp;
S3: k = c1*a1->q*yt;
S4: a1->VP += k*(*nodelist)[inode].q010;
```

In this example, a1->VP has a possible dependence with (*nodelist)[inode].q100, (*nodelist)[inode].sqp, and (*nodelist)[inode].q010. The CDDA detects no dependences among a1->VP, a1->dpx and a1->q based on the results of rule-based alias analysis in ORC. In the example, after copy propagation on a1->VP, the *store* to a1->VP in S1 is speculative dead because it is overwritten in S4 and (*nodelist)[inode].sqp in S2, (*nodelist)[inode].q010 in S4 may need the result of the *store*. Deleting the *store* in S1 contributes very little to the performance gain because the *stores* can be issued for free and overlapped with the *load* of a1->dpx. Similar examples can be found in benchmark *twolf*. For all other benchmarks, *speculative dead store elimination* and *copy propagation* have very few optimization opportunities.

DSCM may increase code size by generating recovery code. IA64 is an EPIC architecture and NOP instruction may be generated by compiler to occupy an instruction slot if there are not enough instructions in one cycle. Our verification and check instructions may use the instruction slot which is originally occupied by a NOP. From our experiment, we observed on average a 5% code expansion.

7. Conclusions

In this paper, a general data speculation compiler framework is presented to enable data speculation in non-speculative optimizations such as redundancy elimination, dead store elimination, copy propagation and code scheduling. The key idea in the framework is to use *data speculative code motion* (DSCM) to move an instruction to a position that would trigger a non-speculative optimization. During DSCM, recovery code will be generated to guarantee the correctness of the code motion. After DSCM, the optimization becomes the same as the non-speculative one. In the proposed framework, a shared *speculative data dependence analysis* (SDDA) is used to hide low probability dependences from optimizations. Hence, no changes are needed to those optimizations to identify speculative optimization opportunities. The SDDA and DSCM can be shared by all optimizations in the proposed framework. These two advantages greatly simplify the task of adopting data speculation into those optimizations. We have implemented the recovery code generation using the *advanced load* and *predication* instructions in IA64 [3]. Our results show that the proposed framework can be efficiently implemented in IA64 and achieve significant speedups for some benchmarks.

8. Acknowledgements

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

9. References:

[1] B.-C. Cheng, W.-M. Hwu. “Modular Interprocedural Pointer Analysis Using Access Paths: Design, Implementation, and Valuation”, in Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), SIGPLAN Notices 35(5), pages 57--69, June 2000

[2] Open Research Compiler for Itanium Processors, http://ipf-orc.sourceforge.net, Jan 2003.

[3] Intel Corp., IA64 Application Developer’s Architecture Guide, http://developer.intel.com/design/ia64/downloads/adag.htm

[4] Fred Chow, Sun Chan, Shin-Ming Liu, Raymond Lo, and Mark Streich. “Effective Representation of Aliases and Indirect Memory Operations in SSA Form”, in Proceedings of the International Conference on Compiler Construction (CC), pages 253--267, 1996.

[5] Robert Kennedy, Sun Chan, Shin-Ming Liu, Raymond Lo, Peng Tu, and Fred Chow. “Partial Redundancy Elimination in SSA Form”, ACM Transactions on Programming Languages and Systems, 21(3) pages 627--676, 1999.

[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 Conference on Programming Language Design and Implementation (PLDI), pages 26--37, June 1998.

[7] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck, "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph," in ACM Transactions on Programming Languages and Systems, pages 451--490 October 1991.

[8] Intel Corp., Itanium Processor Micro-architecture Reference, http://developer.intel.com/design/ia64/donwloads/245473.htm, March 2000.

[9] S. Eranian, “Pfmon Performance Monitoring Tool.” ftp://ftp.hpl.hp.com/pub/linux-ia64

[10] U. Banerjee. Dependence Analysis for Supercomputing. Kluwer Academic Publishers, Boston, MA, 1988.

[11] R. D.-C. Ju, K. Nomura, U. Mahadevan, and L.-C. Wu, “A Unified Compiler Framework for Control and Data Speculation”, in Proceedings of 2000 Int’l Conf. On

Parallel Architectures and Compilation Techniques (PACT), pages 157--168, October 2000

[12] 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 Proceedings of the Six International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pages 183--93, 1994.

[13] Matthew Postiff, David Greene, Greene and Trevor Mudge. “The Store-Load Address Table and Speculative Register Promotion”, in Proeedings of the 33rd Annual Intl. Symp. on Microarchitecture (MICRO), pages 235--244, December 2000.

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

[15] A. S. Huang, G. Slavenburg, and J. P. Shen. “Speculative Disambiguation: A Compilation Technique for Dynamic Memory Disambiguation”, in Proceedings of the 21st International Symposium on Computer Architecture (ISCA), pages 200--210, April 1994.

[16] J. Lin, T. Chen, W.-C. Hsu, P.C. Yew, D.-C. Ju, T.-F. Ngai, S. Chun, “A Compiler Framework for Speculative Analysis and Optimizations”, in Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Pages 289--299, 2003.

[17] J. Knoop, O. Ruthing, and B. Steffen, "Partial Dead Code Elimination", in Proceedings of ACM SIGPLAN Conf. on Prog. Lang. Design and Implementation (PLDI), pages 147--158, 1994.

[18] Preston Briggs and Keith D. Cooper. “Effective Partial Redundancy Elimination”, in Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), SIGPLAN Notices, 29(6) pages 159--170, June 1994.

[19] Chow, F., Chan, S., Kennedy, R., Liu, S., Lo, R., and Tu, P. “A New Algorithm for Partial Redundancy Elimination Based on SSA form”, in Proceeding of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pages 273--286, 1997.

[20] Chen, T., Lin, J., Dai, X., Hsu, W.-C., and Yew, P.-C., “Data Dependence Profiling for Speculative Optimization”, in Proceedings of the 13th International Conference on Compiler Construction (CC), pages 57--72, March 2004.

[21] Bjarne Steensgaard, “points-to Analysis in Almost Linear Time”, POPL, pages 32--41, 1996.

[22] W.Y. Chen, S.A. Mahlke, W.M. Hwu, T. Kiyohara, and P.P. Chang, “Tolerating Data Access Latency with Register Preloading”, in Proceedings of the 6th International Conference on Supercomputing, July 1992, pp. 104--113.