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 04-012

A General Compiler Framework for Data Speculation Using DSCM

Xiaoru Dai, Wei-chung Hsu, and Pen-chung Yew

March 01, 2004

A General Compiler Framework for Data Speculation Using DSCM

Xiaoru Dai
Department of Computer Science
University of Minnesota
Minnesota, MN 55455
dai@cs.umn.edu

Wei-Chung Hsu
Department of Computer Science
University of Minnesota
Minnesota, MN 55455
hsu@cs.umn.edu

Pen-Chung Yew
Department of Computer Science
University of Minnesota
Minnesota, MN 55455
yew@cs.umn.edu

ABSTRACT

Getting precise alias information in a language that allows pointers, such as C, is expensive. One reason is that alias analysis should generate conservative (safe) alias information. Alias analysis assumes possible aliases when it can't prove there are no aliases. The conservative alias information may greatly affect compiler optimizations. In this paper, we present a general framework to allow speculative (unsafe) alias information to be used in several compiler optimizations such as redundancy elimination, copy propagation, dead store elimination and code scheduler. Speculative alias information is not guaranteed to be correct. Run-time verification and recovery code are generated to guarantee the correctness of the optimizations that use speculative alias information. In the framework, the key idea is to use data speculative code motion (DSCM) to move the instruction that will be optimized away to the instruction that causes the optimization. During DSCM, verification and recovery code are generated. After the DSCM, the optimization becomes non-speculative and the moved instruction can be optimized away.

1. Introduction

Alias analysis plays an important role in compilers of language, such as C, which allows pointers. Precision of alias analysis can greatly affect the quality of optimized code. More than eighty papers have been published in alias analysis. It is still quite hard to get precise alias information in an efficient way [1]. One basic requirement of alias analysis is safety. If alias analysis can't prove there is no alias between two memory references, it should assume there is possible alias between the two memory references. Otherwise, the results of the alias analysis may be wrong. Safety requirement may reduce precision of alias analysis dramatically. Most possible aliases generated by alias analysis do not happen at all. With safety requirement, increasing precision of alias analysis becomes quite expensive which may make alias analysis impractical.

Precise alias information is important in compiler optimizations. In Figure 1, two motivation examples show how imprecise alias information may affect compiler optimizations. In the left part of Figure 1, lines represent possible aliases. The possible alias between p and q (line 1) prevents possible redundancy elimination on *q in s3. The possible alias between p and r (line 2) prevents possible copy propagation on *p in s5. The possible alias between p and r (line 3) prevents possible dead store elimination in s4. In this example, three compiler optimizations (redundancy elimination, copy propagation, dead store elimination) are inhibited by possible aliases. If we ignore these possible aliases, it is quite easy for these optimizations to find these optimization opportunities.

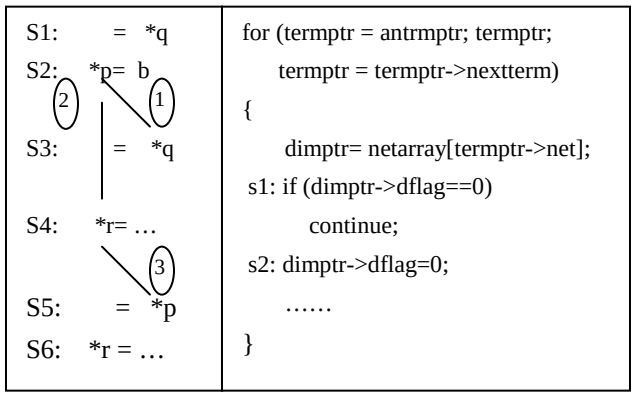

Figure 1: Motivation Examples

In the right part of Figure 1, the load (dimptr->dflag) in s1 does alias with the store (dimptr->dflag) in s2. When the loop is unrolled, the load in s1 can't be scheduled before the store in s2 from the previous iteration because of the alias. Since this alias happens infrequently at run-time, it may be still profitable to schedule the load in s1 before the store in s2 from the previous iteration to hide the load latency if we can recover when alias happens.

Getting precise alias information is expensive because where a pointer may point to is hard to know at compile time especially when it points to heap objects which don't have a name. At run time, where a pointer points to can be easily figured out since the value of the pointer is available. It is easy to verify alias relation between two memory references at run time. Normally, only a simple compare instruction is needed. More and more instructions can be issued in a single cycle on modern processors. For example, IA64 processor [10] can issue six instructions in one cycle. If there are not enough independent instructions, some resources are wasted. Run-time verification of alias relation may be a good candidate to use these otherwise wasted resources.

A compiler optimization normally has two parts, analysis part and code transformation part. The analysis part finds optimization opportunities based on IR and information such as alias information. Code transformation part modifies IR to generate the optimized code.

The above observations inspire our data speculation framework. In this paper, data speculation means speculation on data dependence. In our framework, a speculative alias analysis (SAA) is used to find highly possible aliases and ignore all others even when it can't prove there is definitely no alias. The results of alias or data dependence profile [26] can also be used in our framework. The analysis part of an optimization does analysis based on the results of SAA instead of conservative alias information to find more optimization opportunities. The code transformation part of an optimization which is relatively simpler than analysis part is modified to generate verification and recovery code. Verification code verifies alias relations that are ignored by SAA and optimizations. When verification code finds one alias relation that is ignored by SAA and optimizations does happen at run-time, recovery code is called to recover program to correct status. We use data speculative code motion (DSCM) which refers to code motion with potential data dependence violation to model the problem of verification and recovery code generation. According to the type of the moved instruction (memory read or write), direction of the motion (up or down) and the type of the crossed instruction (memory read or write), there are six cases of DSCM. The verification and recovery code generation for one case may be shared by different optimizations.

The rest of the paper is organized as follows. In section 2 we describe SAA. In section 3 data speculative code motion and it's verification and recovery code generation are discussed. In section 4 implementation of our framework is showed. In section 5 we measure the performance of our framework on SPEC CPU2000 benchmark. In section 6 we describe related work. In section 7 we present our conclusions.

2. Speculative Alias Analysis

Alias analysis tries to determine when two memory references refer to the same memory location [1]. Alias analysis has to make conservative assumption about alias relation because of the safety requirement. Our SAA is different from traditional alias analysis we mentioned above. If SAA doesn't think there is high possibility that two memory references are aliased, SAA just assumes there is no alias between the two memory references. SAA doesn't guarantee the correctness of the results. Without the safety requirement, SAA can use some aggressive heuristic rules and becomes very simple.

We use access paths [3] to represent memory references. An access path (AP) of a memory reference is a non-empty string that consists of variable name, field name of structure, dereferences, and so on, to reach the memory location of the memory reference. For example, the access path of memory reference of a->b->c can be represented as "a*.b*.c" (* represents dereference). We use similar rules as in [3] to generate access paths for memory references. One main difference is that our access path can include index of array references which cannot be in their access path because it is not safe. In alias analysis, heap objects normally are named by where they are allocated. It is hard to distinguish two heap objects if they are allocated at the same place. We name heap objects when they are accessed using the access path. This gives us an opportunity to distinguish heap objects even when they are allocated at the same place.

Heuristic rules are used to find highly likely aliased memory references. Two heuristic rules and when they should apply are showed below.

- Rule 1: Apply when two memory references have same access path. If variables on this access path are not explicitly changed between the two memory references, the two memory references are aliased. Otherwise they are not aliased.
- Rule 2: Apply when two memory references have different access path. Two memory references are not aliased if their access paths are different.

In SAA, for each memory reference in a procedure, access path is generated. After that, alias relations between memory references are generated based on the two heuristic rules using the access paths of memory references.

3. Data Speculative Code Motion

Data speculative code motion (DSCM) refers to the code motion with potential data dependence violation. Here we limit data dependence to data dependence caused by memory references since data dependence caused by registers does not need speculation. According to the type of the moved instruction (memory read or write), the direction of the motion (up or down) and the type of the crossed instruction (memory read or write), we classify DSCM into six cases that are showed in Figure 2. In each case, the dotted arrow lines represent the direction of the code motion. The solid arrow lines represent control flow. 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.

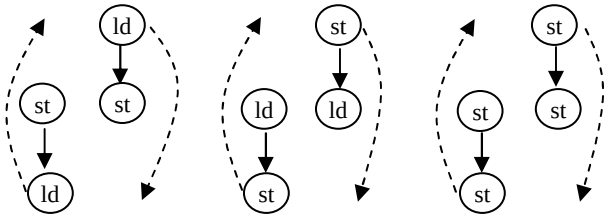

Figure 2: Six cases of DSCM

	moved inst	crossed inst	direction	may violate dependence
case 1	ld	st	up	true dependence
case 2	ld	st	down	anti dependence
case 3	st	ld	up	anti dependence
case 4	st	ld	down	true dependence
case 5	st	st	up	output dependence
case 6	st	st	down	output dependence

Table 1: summary of six cases of DSCM

From the view of code motion, it's equivalent for case 1 and case 4, case 2 and case 3, case 5 and case 6. We consider that they are different because their verification and recovery code generation are different. We only generate verification and recovery code for the moved instruction. The crossed instruction is not changed. Moving a ld up or down across a potential aliased ld may violate

input dependence [12]. Normally, violation of input dependence would not affect the correctness of the program. So the input dependence violation is not considered in our framework.

3.1 Verification and Recovery Code Generation

We need verification code to find out if one particular DSCM is correct or not. When verification code finds out one DSCM is wrong, recovery code should be executed to recover program to correct status. DSCM may violate possible data dependences. Comparing the memory reference address of the moved ld or st and the memory reference address of the crossed ld or st can tell us if data dependence violation happens or not. What should be in a recovery code and when recovery code should be executed depend on the case of DSCM. Normally, the recovery code should include the moved instruction. If the direction of DSCM is up, we can execute the recovery code at the original place of the moved instruction. If the direction of DSCM is down, we may execute recovery code as soon as we find DSCM is not correct. There are four parts to support one particular DSCM.

- Flag initialization: set a flag to indicate that DSCM is successful.
- Verification instruction: by comparing the addresses of the moved memory reference and the crossed memory reference, verify if DSCM is successful. If it is not, the flag set in initialization is cleared.
- Check instruction: check if DSCM is successful. If it is not, execute recovery code.
- Recovery code: a copy of the moved instruction.

We select case 1, case 4 and case 6 of DSCM as examples to show how to support DSCM. The flag initialization, verification instructions, check instructions and recovery code are showed in Figure 3.

Case 1:	Case 4:	Case 6:
Before DSCM:	Before DSCM:	Before DSCM:
st [r1] = r2	st [r1] = r2	st [r1] = r2
ld r3 =[r4]	ld r3 = [r4]	st [r3] = r4
After DSCM:	After DSCM:	After DSCM:
s1: flag =1	s1: flag =1	s1: flag =1
s2: ld r3 = [r4]	s2: if (r1 == r4)	s2: if (r1 == r3)
s3: if (r1 == r4)	flag =0	flag =0
flag =0	s3: if (flag==0)	s3: st [r3] = r4
s4: st [r1]= r2	s4: st[r1] = r2	s4: if (flag == 1)
s5: if (flag == 0)	s5: ld r3 = [r4]	s5: st [r1] = r2
s6: ld r3=[r4]	s6: if (flag == 1)	
	s7: st [r1] = r2	

Figure 3: support of DSCM, case 1, 4 and 6

Case 1 moves a ld up across a potential aliased st. In the result code, s1 sets a flag to 1 to indicate the code motion is successful. S2 executes the ld that is now before the st. S3 is to verify if DSCM successful or not. If the address (r4) of the ld is equal to the address (r1) of the st, then the code motion violates true data dependence and the flag is cleared to indicate DSCM failure. S4 executes the st which is not changed. S5 is a check instruction to check the flag. If the flag is 0 that indicates DSCM failure, then recovery code S6 is executed. In this example, the recovery code is the original ld.

Case 4 is similar as 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 DSCM fails, the following ld needs the result of the st. The check instruction (s3) can be optimized away since flag==0 is equivalent to r1==r4. If DSCM fails, there is one possible optimization. We may skip the following ld because we know that the result of the ld should be value r2. This optimization is not included in our framework because we want to keep the crossed ld unchanged to simplify our framework. In case 4 DSCM, the moved instruction (s7) is guided by a condition (s6). Whether we should executed the moved instruction or not depends on the status of DSCM. If DSCM succeeds, we execute the moved instruction. Otherwise, we do not.

In case 6, we do not need check instruction and recovery code. If DSCM fails, we know that the crossed st overwrites the moved st. The moved st (s5) is guided by a condition (s4) which is the same as case 4 DSCM.

3.2 Optimizations Using the Results of SAA

When an optimization is based on the results of SAA, we call this optimization a data speculative optimization because the result of the optimization is no longer always correct because of possible data dependence violation. We use four compiler optimizations to show how to relate the problem of generating code to guarantee the correctness a data speculative optimization to the six cases of DSCM. The idea is to use DSCM to move the instruction that will be optimized away to the instruction that causes the optimization. During the DSCM, verification and recovery code are generated. After the DSCM, the optimization becomes non-speculative and the moved instruction can be optimized away.

3.2.1 Data Speculative Dead Store Elimination

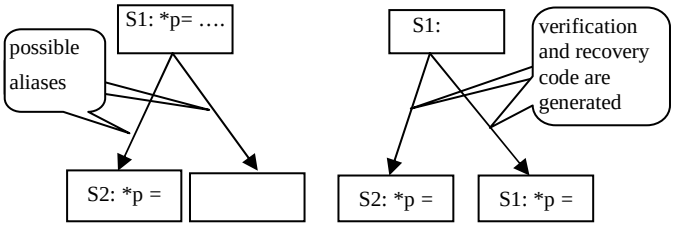

Figure 4: data speculative dead store elimination

Dead store elimination [23] [8] eliminates unnecessary stores of which the results are not used. For the example in the left part of Figure 4, if we ignore all the possible aliases, the store in s1 is dead when the left path is taken. Traditional dead store elimination wouldn't consider the store in s1 dead because the result of the store may be used if possible aliases happen. In data speculative dead store elimination, the store in s1 is dead because all possible aliases are ignored. In this example, the store in s1 is the instruction that will be optimized away. The store in s2 is the instruction that causes the optimization. We use DSCM to move

s1 as showed in Figure 5. The dotted lines show the direction of the code motion. After the DSCM, the optimization becomes simple non-speculative dead store elimination and the result is showed in the right part of Figure 4. The correctness of the data speculative dead store optimization is equal to the correctness of the DSCM during which verification and recovery code are generated. In the DSCM, a st is moved down which may require case 4 and case 6 of DSCM.

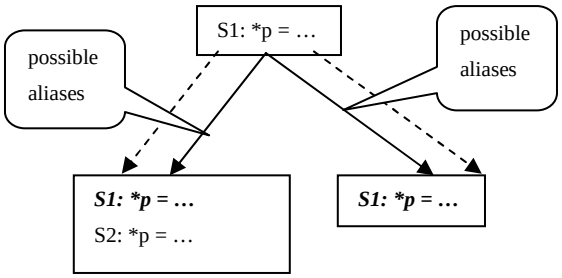

Figure 5: DSCM for data speculative dead store elimination

3.2.2 Data Speculative Partial Redundancy Elimination

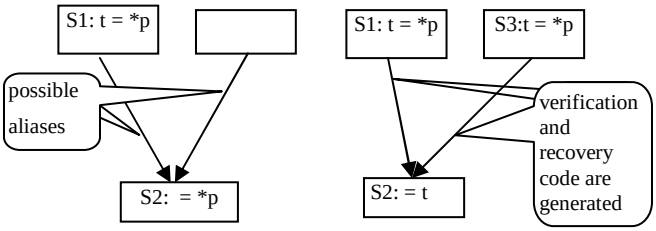

Figure 6: data speculative partial redundancy elimination

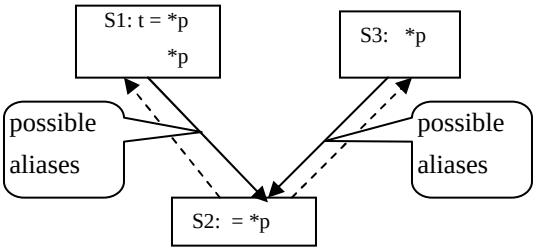

Figure 7: DSCM for data speculative partial redundancy elimination

Partial redundancy elimination [24][7][25] can eliminate loads or computations of which the results are already available. For the example in the left part of Figure 6, if we ignore all possible aliases, the result of the load (*p) in s2 is available when the left path is taken. Traditional partial redundancy elimination can't eliminate the load in s2 because there are possible aliases and *p may be modified after s1. In data speculative partial redundancy elimination, after all possible aliases are ignored, the load (*p) in s2 is the instruction that will be optimized away. The load (*p) in s1 is the instruction that causes the optimization. We use DSCM to move the load in s2 as showed in Figure 7. The dotted lines represent the direction of the DSCM during which verification and recovery code are generated. After the DSCM, the optimization becomes simple redundancy elimination and the result is showed in the right part of Figure 6. The correctness of the speculative optimization is equal to the correctness of the DSCM which requires case 1 of DSCM.

3.2.3 Data Speculative Copy Propagation

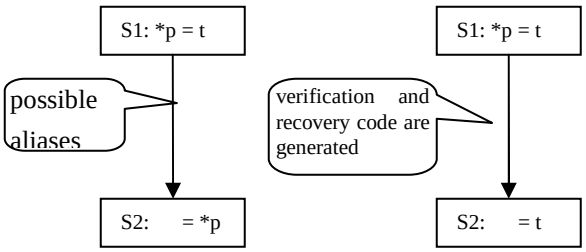

Figure 8: data speculative copy propagation

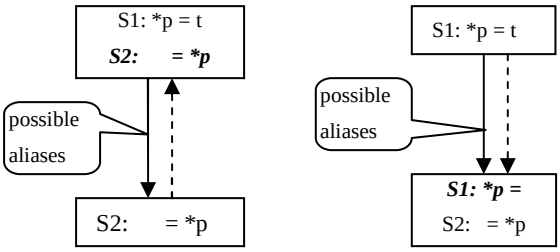

Figure 9: DSCM for data speculative copy propagation

Copy propagation can replace a load if a store to the same memory location can be found before the load. For the example in the left part of Figure 8, if all possible aliased are ignored, the load (*p) in s2 will get the result of the store in s1 which is t. So the load in s2 can be replaced by value t. Traditional copy propagation can't do this because of possible aliases. *p may be changed after s1. In data speculative copy propagation, after all possible aliases are ignored, the load (*p) in s2 can be optimized away. The load in s2 is the instruction that will be optimized away. The store in s1 is the instruction that causes the optimization. We use DSCM to move the load (*p) in s2 as showed in the left part of Figure 9 or move the store in s1 as showed in the right part of Figure 9. After the DSCM, copy propagation becomes simple because the store and the load are near each other now. The result of the optimization is showed in the right part of Figure 8. Moving *p in s2 up requires case 1 DSCM. Moving *p in s1 down requires case 4 and 6 DSCM. During DSCM, verification and recovery code are generated. If we use case 1 DSCM, the verification and recovery code generation can be shared with data speculative partial redundancy elimination. If we use case 4 and 6 DSCM, the verification and recovery code generation can be shared with data speculative dead store elimination.

3.2.4 Data Speculative Code Schedule

The code transformation of a code scheduler is similar as a code motion. Starting from cycle 0, scheduler selects the best candidate

to issue in this cycle. We can consider the process as selecting the candidate and moving it from the original place to cycle 0. When cycle 0 is full or no instructions can be issued in the cycle, code scheduler advances the cycle to cycle 1 and repeats the process until all instructions are scheduled. Since DSCM is one kind of code motion, it is quite nature to map the code transformation of code scheduler to DSCM. There is one thing we need mention. Code scheduler may move any instructions and, as far as we discussed, DSCM only moves load or store. When an instruction is moved across a check instruction of a memory instruction that it directly or indirectly depends on, the moved instruction should be added to the recovery code of the check instruction. An example is showed in the next section.

4. Implementation of the Framework

The data speculation framework is implemented in Open Research Compiler (ORC) [4]. ORC generates code for IA-64 [5] [10]. The performance of our framework is highly dependent on application characteristics and the implementation. We use two architectural features, speculation and predication, which are available on IA-64 to implement our DSCM support. Compiler optimizations such as partial redundancy elimination, dead store elimination, copy propagation, code scheduler have been implemented in ORC. In Figure 10, structure of related parts of ORC is showed. ORC uses WHIRL (Winning Hierarchical Intermediate Representation Language) as it's IR.

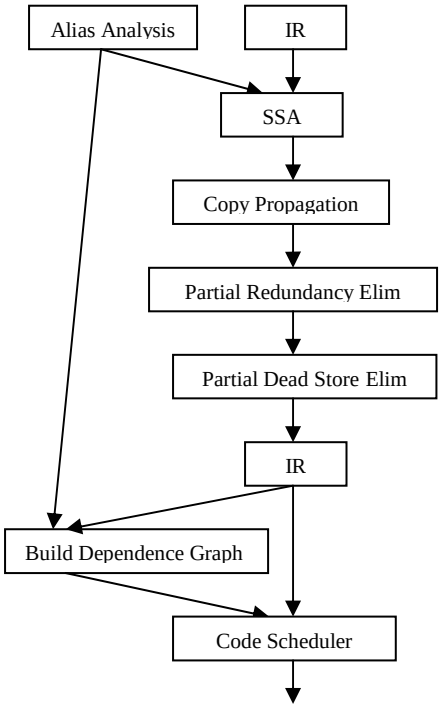

Figure 10: Structure of Related Parts of ORC

Alias analysis is an important part in ORC. Most optimizations in ORC need the results of alias analysis. Alias analysis in ORC has two parts. First part is a points-to analysis which is based on a non-standard type system [2]. Second part is a rule based alias analysis. Base rule which refers to that two memory references are not aliased if their base is different, offset rule which refers to that two memory references are not aliased if their base is the same and their memory ranges define by the offset and size fields does not overlap, type rule which refers to 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, and so on are included in rule based alias analysis. Based on the results of alias analysis and IR, static single assignment (SSA) form is created [9][6]. Copy propagation, partial redundancy elimination [7] and partial dead code elimination [8] are applied in SSA form. After these optimizations, SSA form is transferred back to WHIRL. Before ORC schedules code, WHIRL is transferred to OP (machine-level operation) which is not showed in Figure 10. A dependence graph is built for the code scheduler. During the construction of the dependence graph, the results of alias analysis are used. Based on the dependence graph and resource constrictions, code is scheduled.

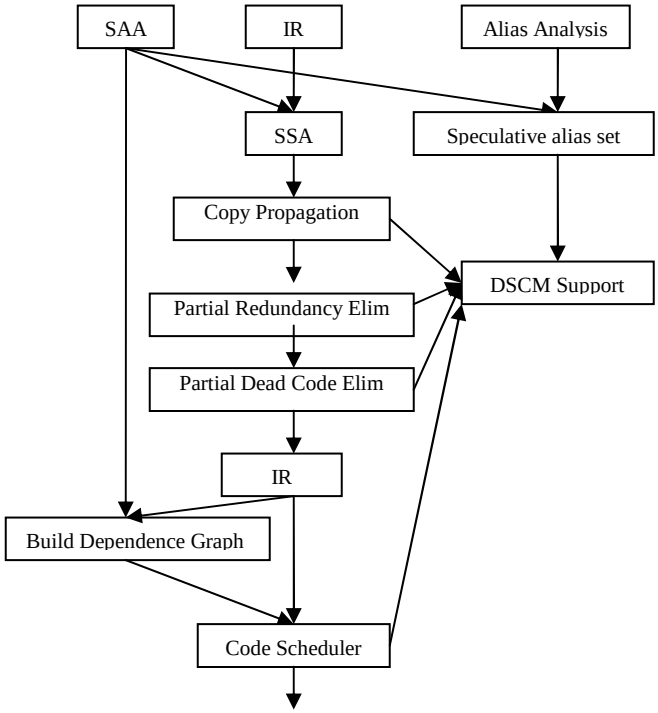

Figure 11: Structure of Data Speculation Framework

Figure 11 shows the structure of our framework. Original alias analysis (AA) is replaced by SAA. Now the construction of SSA and dependence graph is based on SAA. We compare the results of AA and SAA. If an alias relation is in results of SAA and is not in the results of AA, we know SAA is wrong about this alias relation. We delete this alias relation from the results of SAA. If an alias relation is in results of AA and is not in the results of SAA, we know SAA is speculative on this alias relation. We keep this alias relation in a speculative alias set. DSCM support can use this set to find out quickly if one code motion needs speculation support or not. Only DSCM support need access the speculative alias set. In our framework, we re-use the original optimizations. The code transformation part of the optimizations is modified to notify DSCM support what kind of code motion it will need. For one code transformation, information about which instruction is moved and where is the destination of the motion is given to

DSCM support. Based on this information, DSCM support decides the case of DSCM and generates verification and recovery code. After this, the moved instructions may be optimized away by the optimizations.

4.1 DSCM Support

In our implementation, DSCM support can generate verification and recovery code for case 1, 4 and 6 DSCM. Case 2, 3 and 5 are not implemented.

4.1.1 Case 1 DSCM

IA-64 architecture includes instructions ld.a (advanced load) and chk.a (advanced load check) for case 1 DSCM [5]. Ld.a is like a normal load but also stores the destination register number and memory address it loads from in ALAT (Advanced Load Address Table). A store instruction will invalidate any entry in ALAT that has same memory address as the store. Chk.a checks the entry set by a ld.a and will trigger a branch to invoke recovery code if the entry is invalid. One example is showed in Figure 12. Recovery code for case 1 DSCM consists of the moved load instruction. When computation instruction which directly or indirectly depends on the moved instruction is moved across the corresponding chk.a instruction, this instruction is also added to recovery code.

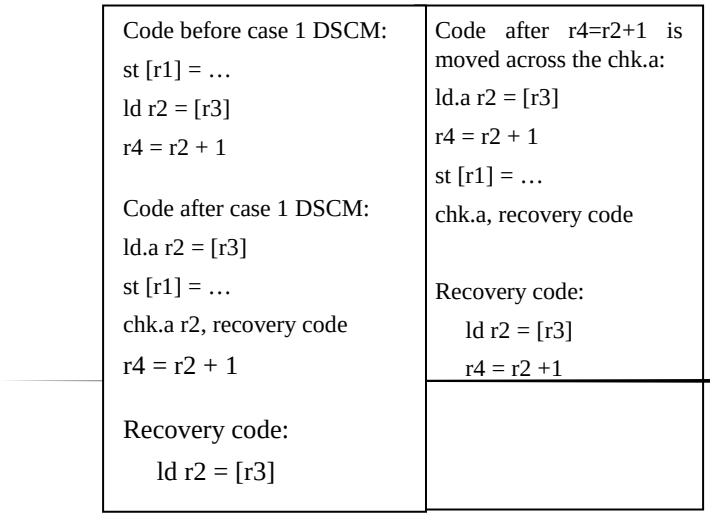

Figure 12: verification and recovery code for case 1

4.1.2 Case 4 and case 6

We use predicate register on IA-64 to hold the result of verification instructions. There are 64 predicate registers on IA64. On IA64, a compare instruction can set two predicate registers 1 or 0 at the same time. For example, p1, p2 = cmp.ne 2, 3 will set predicate register p1 to 1 and p2 to 0. Instructions on IA-64 are guided by predicate register. If the predicate register of an instruction is 0, then this instruction is changed to a NOP during execution. We use this feature (predication) to generate more compact code for verification and recovery code. For case 6, no recovery code is needed because when alias does happen during execution time, we know the moved st is killed by the crossed st. The result code is showed in the right part of Figure 13. For case 4, the result code is showed in the left part of Figure 13.

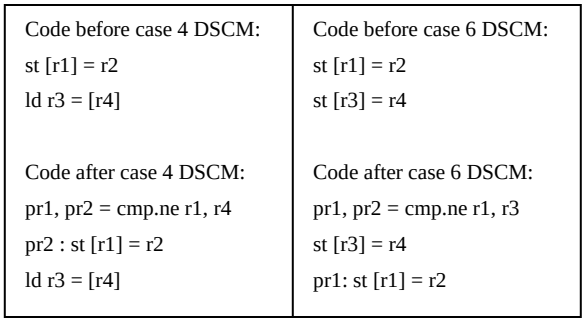

Figure 13: verification and recovery code for case 4 and 6

4.1.3 DSCM crosses multiple aliased sts or lds

As far as we discussed, DSCM only moves a ld or st across one aliased ld or st. In order to move a ld or st across multiple aliased lds or sts, we need a flag to indicate if a verification instruction or a check instruction should be skipped. After recovery code is executed, all the following verification instructions and check instructions should be ignored. For case 1, we only have one check instruction after which there is no more verification and check instructions. For case 4 and case 6, we need generate check instruction for every crossed aliased memory reference. In the following example in Figure 14, we move the first st across two sts and two lds. All verification instructions except the first one are guided by predicate register pr1 which is set to 0 when DSCM fails. All check instructions are guided by predicate register pr2 which is set to 1 when DSCM fails. After recovery code is executed, pr2 is set to 0. When pr1 and pr2 are all 0, all the following verification instructions and check instructions are ignored.

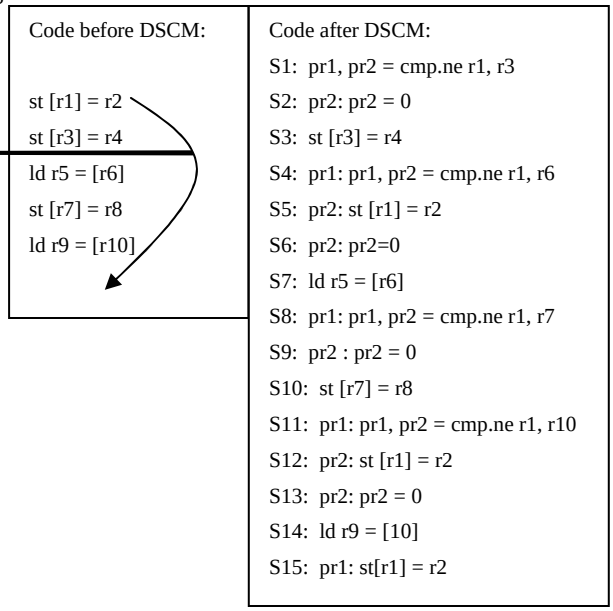

Figure 14: A st is moved across multiple lds and sts

In the example, s2, s6, s9, s13 are the 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.

5. Experimental Results

We use seven benchmarks from SPEC CINT2000 and four benchmarks which are written in C from SPEC CFP2000 to measure the effectiveness of the framework. The measurements were performed on Itanium2 processor [[10]. We measure the speedup for each benchmark relative to its base case, which is optimized with –O3 option of ORC. In the following graph, we list the percentages of run-time performance (cpu cycle) improvements. Pfmon [11] is used to collect performance data. All benchmarks run with reference input.

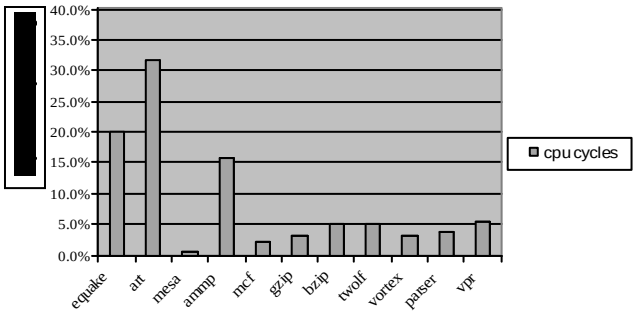

Figure 15: improvement of cpu cycle

In Figure 15, improvement of performance (reduction in cpu cycle) is showed. There are three benchmarks (equake, art, ammp) having performance improvement above 15%. They are all from SPEC CFP2000. Floating memory reference dominates the memory references in the three benchmarks. Data speculative optimization opportunities are good in the time-consuming part of the three benchmarks. Most performance improvement comes from data speculative redundancy elimination and code scheduler. Data speculative redundancy elimination can delete redundancy load and computation. Data speculative code scheduler can allow more overlap between instructions to hide load and computation latency. For equake, speculative redundancy elimination in procedure main and smvp contributes about 7% improvement. Data speculative code scheduler contributes about 13% improvement. For art, about 5% improvement is from data speculative redundancy elimination. About 27% improvement is from data speculative code scheduler. For ammp, about 5% improvement is from data speculative redundancy elimination. About 10% improvement is from data speculative code scheduler. For all other benchmarks, the improvement ranges from 0.5% to 5%. The performance improvement of data speculative redundancy elimination is not good in integer benchmarks because the integer load latency is only 1 cycle when the load hits cache. If a load results a cache miss, normally, this load is not redundant. For data speculative code scheduler, the performance improvement is also not good. Link list traversal dominates memory references in the most integer benchmarks. It's hard for code scheduler to overlap memory references to two elements in a link list because of register data dependence.

Speculation fail rate, load number reduction and store number reduction are showed in the following table. We compute speculation fail rate as number of failed check instructions divided by number of advanced load since speculation opportunities in dead code elimination and copy propagation is very little. We use pfmon to collect these numbers.

Benchmark	Speculation fail rate	Load number Reduction	Store number Reduction
equake	0.20%	31.6%	<0.01%
art	0.02%	10.0%	<0.01%
mesa	<0.01%	<0.01%	0.12%
ammp	8.20%	17.2%	35.6%
mcf	1.50%	4.6%	<0.01%
bzip	0.50%	1.3%	<0.01%
twolf	0.18%	5.8%	9.28%
vortex	0.17%	1.5%	<0.01%
parser	0.67%	0.2%	0.58%
vpr	0.02%	6.5%	0.31%
gzip	0.10%	0.6%	<0.01%

The penalty of recovery code is very high because, normally, recovery code is not in instruction cache and at least we have one branch mis-prediction for the check instruction. The speculation fail rate for most benchmarks is below 1% that avoids high penalty of recovery code. Ammp has a high fail rate about 8% because there are many stores between advanced load and it's corresponding check instruction which results high possibility of false conflict in ALAT because ALAT only keeps partial address of an advanced load. When we change check instructions to NOPs, another 7% improvement of performance can be achieved.

The load number reduction indicates the opportunities of data speculative redundancy elimination. From the data, floating benchmarks have larger load number reduction. From my experience, floating benchmarks are less optimized by the programmers. Normally, larger load number reduction results bigger improvement of performance.

For store number reduction, only two benchmarks (ammp, twolf) have large store number reduction. In our framework, data speculative dead store elimination and copy propagation don't give much performance improvement. We checked the benchmark ammp for the reason that high reduction in store number doesn't result high performance improvement. In the important procedure of ammp, we found opportunities for speculative dead store elimination. An example code is showed 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 the example, a1->VP is aliased with (*nodelist)[inode].q100, (*nodelist)[inode].sqp, and (*nodelist([inode].q010. The rule based alias analysis in ORC can detect there is no alias among a1->VP, a1->dpx and a1->q. Since all stores are type float, pointers which are 64 bits integer are not changed based on the type rule. In the example, after copy propagation on a1->VP, the store to a1->VP in s1 is speculative dead store 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 performance gain because the store can overlap with the

load (a1->dpx) in s2. Similar example can be found in benchmark twolf. For all other benchmarks, data speculative dead store elimination and copy propagation have very limited optimization opportunities.

6. Related Work

Speculation is quite common in compiler and computer architecture. Branch prediction [13] can let processor fetch instructions before the condition of the branch is known. If the prediction turns out to be wrong, the processor has to discard all instructions that were fetched based on this prediction. Data pre-fetch is another example of speculation. Compiler can generate pre-fetch instructions to bring data from memory to cache to hide load latency. In data pre-fetch, no recovery action is required. Pre-fetching unused data into cache may hurt performance. But it will not affect the correctness of the program.

Normally, the results of speculative alias analysis cannot be used in compiler optimizations since the results may be wrong. In [14] [20] [21], the results of their speculative alias analysis are used in their speculative multithreading compilers to generate better thread. They rely on the recovery ability of multithreading processor to guarantee the correctness.

Speculative optimizations on single processor have been studied in several papers [15] [16] [17] [18] [19] [22]. The key idea is to provide some kind of check and recovery mechanism to guarantee the correctness of speculative optimizations. In these papers, hardware support for speculative optimizations is studied and used. In [15], a compiler framework for control and data speculation in a code scheduler is presented. Three main tasks in scheduling speculative instructions: marking speculative dependence edges, selecting speculative instructions as scheduling candidates, and check insertion and DAG update are fully integrated with the rest of the instruction scheduling phase. In [22], a speculative SSA form incorporating data speculation information is created for SSAPRE [7] which takes SSA form as it's input IR. SSAPRE is modified in [22] to consider speculative SSA form as input IR. In [15] [22], speculative information is explicitly presented in IR and optimizations are modified to handle the speculative information. Our data speculation framework is designed to easily change traditional non-speculative optimizations to speculative optimizations. Major modification on optimizations is in their code transformation part. Speculative instructions are not treated differently from ordinary instructions in the optimization analysis part. Further more, the verification and recovery code generation is shared among speculative optimizations.

7. Conclusion

In this paper, a general compiler framework is presented to allow traditional optimizations such as partial redundancy elimination, partial dead store elimination, copy propagation, code scheduler to use speculative alias information instead of conservative alias information. The key idea in the framework is to use data speculative code motion to move an instruction that will be optimized to the instruction that causes the optimization. During data speculative code motion, verification and recovery code are generated to guarantee the correctness of the optimizations that now are based on speculative alias information. After data speculative code motion, the optimization becomes trivial. We further classify data speculative code motion into 6 cases. In this paper, the verification and recovery code generation for case 1, 4, 6 are implemented using data speculation and prediction features on IA64 [5]. Case 1 can be shared among data speculative partial redundancy elimination, data speculative code scheduler, data speculative copy propagation. Case 4 and 6 can be shared by data speculative dead store elimination, data speculative copy propagation.

8. REFERENCES

[1] Michael Hind, pointer Analysis: "Haven't We Solved This Problem Yet?", PASTE, 2001

[2] Bjarne Steensgaard, "points-to Analysis in Almost Linear Time", POPL, 1996

[3] B.-C. Cheng, W. mei Hwu. "Modular interprocedural pointer analysis using access paths: Design, implementation, and valuation." In Proceedings of the ACM SIGPLAN'00 Conference on Programming Language Design and Implementation, pages 57-69, 18-21 June. SIGPLAN Notices 35(5), June 2000

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

[5] Intel Corp., IA-64 Application Developer's Architecture Guide, http://developer.intel.com/design/ia64/downloads/adag.htm

[6] 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, pages 253-267, 1996.

[7] 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):627--676, 1999.

[8] 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.

[9] 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. ACM, October 1991.

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

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

[12] U. Banerjee. Dependence Analysis for Supercomputing. Kluwer Academic Pub- lishers, Boston, MA, 1988.

[13] Johnny K. F. Lee and Alan Jay Smith, "Branch prediction strategies and branch target buffer design," IEEE Computer, vol. 17, no. 1, pp. 6-22, January 1984.

[14] Anasua Bhowmik, Manoj Franklin, “A fast approximate interprocedural analysis for speculative multithreading compilers”, Proceeding of the 17th annual international conference on Supercomputing, San Francisco, CA, USA, 2003

[15] 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

[16] 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 Six International Conference on Architectural Support for Programming Languages and Operating Systems, 1994.

[17] 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.

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

[19] A. S. Huang, G. Slavenburg, and J. P. Shen. “Speculative disambiguation: A compilation technique for dynamic memory disambiguation.” In Proceedings of the 21st Symposium on Computer Architecture, pages 200--210, Chicago, Illinois, Apr. 1994. ACM SIGARCH.

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

[21] Peng-Sheng Chen, Ming-Yu Hung, Yuan-Shin Hwang, Roy Dz-Ching Ju, Jeng Kuen Lee, “Compiler support for speculative multithreading architecture with probabilistic points-to analysis”, Proceedings of the 9th ACM SIGPLAN symposium on Principles and practice of parallel programming, San Diego, California, USA, 2003

[22] 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”

[23] J. Knoop, O. Ruthing, and B. Steffen, "Partial Dead Code Elimination," ACM SIGPLAN Conf. on Prog. Lang. Design and Implementation, pages 147-158, 1994.

[24] Preston Briggs and Keith D. Cooper. “Effective partial redundancy elimination”. SIGPLAN Notices, 29(6):159--170, June 1994. Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation.

[25] 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 ’97 Conference on Programming Language Design and Implementation. 273-286, 1997.

[26] Chen, T., Lin, J., Dai, X., Hsu, W.-C., and Yew, P.-C., “Data dependence profiling for speculative optimization”. In 13th International Conference on Compiler Construction, Barcelona, Spain, March 2004.