August G. Reinig

Alias Analysis in the DEC C and DIGITAL C++ Compilers

During alias analysis, the DEC C and DIGITAL C++ compilers use source-level type information to improve the quality of code generated. Without the use of type information, the compilers would have to assume that any assignment through a pointer expression could modify any pointer-aliased object. In contrast, through the use of type information, the compilers can assume that such an assignment can modify only those objects whose type matches that referenced by the pointer.

When two or more address expressions reference the same memory location, these address expressions are aliases for each other. A compiler performs alias analysis to detect which address expressions do not reference the same memory locations. Good alias analysis is essential to the generation of efficient code. Code motion out of loops, common subexpression elimination, allocation of variables to registers, and detection of uninitialized variables all depend upon the compiler knowing which objects a load or a store operation could reference.

Address expressions may be symbol expressions or pointer expressions. In the C and C++ languages, a compiler always knows what object a symbol expression references. The same is not true with pointer expressions. Determining which objects a pointer expression may reference is an ongoing topic of research.

Most of the research in this area focuses on the use of techniques that track which object a pointer expression might point to.[1,2] When these techniques cannot make this determination, they assume that the pointer expression points to any object whose address has been taken. These techniques generally ignore the type information available to the source program. The best techniques perform interprocedural analysis to improve their accuracy. Although effective, the cost of analyzing a complete program can make this analysis impractical.

In contrast, the DEC C and DIGITAL C++ compilers use high-level type information as they perform alias analysis on a routine-by-routine basis. Limiting alias analysis to within a routine reduces its cost, albeit at the cost of reducing its effectiveness.

The use of this type information results in slight improvements in the performance of some standard-conforming C and C++ programs. These improvements come at little expense in terms of compilation time. There is, however, a risk that the use of this type information on nonstandard-conforming C or C++ programs may result in the compiler producing code that exhibits unexpected behavior.

The C and C++ Type Systems

Research available on the use of type information during alias analysis involves languages other than C and C++.[3] Traditionally, C is a weakly typed language. A pointer that references one type may actually point to an object of a different type. For this reason, most alias-analysis techniques ignore type information when analyzing programs written in C.

The ISO Standard for C defines a much stronger typing system.[4] In ISO Standard C, a pointer expression can access an object only if the type referenced by the pointer meets the following criteria:

- It is compatible with the type of the object, ignoring type qualifiers and signedness.
- It is compatible with the type of a member of an aggregate or union or submembers thereof, ignoring type qualifiers and signedness.
- It is the char type.

Thus, in Figure 1, the pointer p can point to A, B, C, or S (through S.sub.m) but not to T or F. The pointer q, being a pointer to char, can refer to any of A, B, C, S, T, or F.

The proposed ISO Standard for C++ defines a similar typing system for C++.[5] The strength of the Standard C and C++ type systems allows the DEC C and DIGITAL C++ compilers to use type information during alias analysis.

Many existing C applications do not conform to the Standard C typing rules. They use cast expressions to circumvent the Standard C type system. To support these applications, the DEC C compiler has a mode whereby it ignores type information during alias analysis. The DIGITAL C++ compiler also has such a mode. This mode exists to support those C++ programmers who circumvent the C++ type system.

```
int A;
signed int const B;
unsigned int volatile C;
struct {
   struct {
       int m;
   } sub;
} S;
struct {
   short z;
} T;
float F;

int *p;
char *q;
```

Figure 1
Code Fragment Associated with the Explanation of the Standard C Aliasing Rules

The Side-effects Package

The DEC C and DIGITAL C++ compilers are GEM compilers.[6] The GEM compiler system includes a highly optimizing back end. This back end uses the GEM data access model to determine which objects a load or a store may access. GEM compiler front ends augment the GEM data access model with a side-effects package, i.e., an alias-analysis package. The side-effects package provides the GEM optimizer additional information about loads and stores using language-specific information otherwise unavailable to the GEM optimizer.

The DEC C and DIGITAL C++ compilers share a common side-effects package. The DEC C and C++ side-effects package

- Determines which symbols, types, and parts thereof a routine references
- Determines the possible side effects of these references
- Answers queries from the GEM optimizer regarding the effects and dependencies of memory accesses

Preserving Memory Reference Information

The DEC C and DIGITAL C++ front ends perform lexical analysis and parsing of the source program, generating a GEM intermediate language (GEM IL) graph representation of the source program.[6] A *tuple* is a node in the GEM IL and represents an operation in the source program.

As the DEC C and DIGITAL C++ front ends generate GEM IL, they annotate each fetch (read) and store (write) tuple with information describing the object being read or written. The front ends annotate fetches and stores of symbols with information about the symbol. They annotate fetches and stores through pointers with information about the type the pointer references. The annotation information includes information describing exactly which bytes of the symbol or type the tuple accesses. This allows the side-effects package to differentiate between access to two different members of a structure.

Arrays Neither the DEC C nor the DIGITAL C++ front end differentiates between accesses to different elements of an array. Both assume that all array accesses are to the first element of the array. The GEM optimizer does extensive analysis of array references.[7] Being flow insensitive, the DEC C and C++ side-effects package can, at best, differentiate between two array references that both use constant indices. The GEM optimizer can do much more.

What the GEM optimizer cannot do, however, is determine that an assignment through a pointer to an int does not change any value in an array of doubles. This is the purpose of the DEC C and C++ side-effects package. Mapping all array accesses to access the first

element of an array does not hinder this purpose and simplifies alias analysis of arrays.

Tuple Annotation Example For the program fragment in Figure 2, the DEC C and DIGITAL C++ front ends generate the annotated tuples displayed in Table 1.

Intraprocedural Effects Analysis

The GEM optimizer makes several optimization passes over a routine. During each optimization pass, the DEC C and C++ side-effects package provides alias analysis information to the GEM optimizer by means of the following procedures:

- Examining each tuple within the routine that references (reads or writes) memory, allocating effects classes that represent the memory that the tuple references
- Performing type-based alias analysis
- Responding to alias-analysis queries from the GEM optimizer

To determine the possible side effects of a memory access, the side-effects package partitions memory into effects classes. An effects class represents all or part of an object. To minimize the number of effects classes under consideration, the side-effects package creates effects classes for only those object regions referenced within the current routine.

Having created effects classes for each referenced object region within the current routine, the side-effects package then associates a signature with each effects class. The signature for an effects class records the possible side effects of referencing the effects class. The side-effects package uses this signature to respond to queries from the GEM optimizer about the effects and dependencies of tuples and symbols within the current routine.

Allocating Effects Classes There are two kinds of effects classes. The first kind represents a region of an individual object. The second kind represents a region of all allocated objects of a particular type. Allocated objects are those created by the `malloc()` function and its relatives or the C++ `new` operator.

As it processes the tuples within a routine, the side-effects package examines the memory reference information associated with the tuple. The side-effects package creates an effects class for each different set of memory reference information it encounters. Two sets of memory reference information are different if they contain different start- or end-offset information or different symbol information.

Two sets of memory reference information that contain different type information are different only if the two types are not effects equivalent. Two types are effects equivalent if they differ only in their signedness or their type qualifiers. The signed int type and the volatile unsigned int type are effects equivalent. An assignment through a pointer to a signed int may change the value of a volatile unsigned int.

Typically, an effects class represents a complete object or an individual member of a structure. An effects class may represent a subregion of the region represented by another effects class. This occurs whenever code references a whole structure as well as individual members of the structure. In the case of unions,

```
struct S {
   int x;
   int y;
} v1, v2;
int i;
double d[3];
struct S *p;

p->x = 3;
v1.y = 3;
v2 = v1;
d[i] = d[0];
```

Figure 2
Code Fragment Associated with Tuple Annotation Example

Table 1
Tuple Annotations

C/C++ Source Expression	Tuple	Annotation Symbol	Annotation Type	Start Byte	End Byte
	Fetch p	p	struct S *	0	7
p->x = 3;	Store p->x	none	struct S	0	3
v1.y = 3	Store v1.y	v1	struct S	4	7
	Fetch v1	v1	struct S	0	7
v2 = v1	Store v2	v2	struct S	0	7
	Fetch d[0]	d	double	0	7
d[i] = d[0]	Fetch i	i	int	0	3
	Store d[i]	d	double	0	7

if two members occupy exactly the same memory locations, a single effects class represents both members.

For the program fragment in Figure 3, the side-effects package creates the effects classes displayed in Table 2.

There is only one effects class for *uip and *ip since uip and ip may point to the same object. There are no effects classes for bytes 0 through 3 of s and struct S as there are no references to s.x or sp->x. By allocating effects classes for only those object regions referenced within the routine, the side-effects package greatly reduces both the number of effects classes and the time required to perform alias analysis.

In the traditional C type system, a pointer expression may point to anything, regardless of type. To represent this, the side-effects package creates exactly one effects class to represent allocated objects. It ignores the type and the start- and end-offset information.

```
struct S {
   int x;
   struct T {
       int y;
       float z;
   } t;
} s;
struct S *sp;
signed int *ip;
unsigned int *uip;
float *fp;

*uip = *ip;
*fp = 2;
sp->t = s.t;
sp->t.y = 2;
s = *sp;
```

Figure 3
Code Fragment Associated with Allocating Effects Classes

Using the traditional C type system, for the program fragment shown in Figure 3, the side-effects package creates the effects classes displayed in Table 3. Here, effects class 7 replaces effects classes 7 through 11 in Table 2. All the differentiation by types disappears.

Effects-class Signatures Having created the effects classes, the side-effects package associates a signature with each effects class. In addition, it associates an effects-class signature with each tuple within the routine and each symbol referenced within the routine.

An effects-class signature records the possible side effects of referencing an effects class. A reference to one effects class may reference another effects class. The effects class for a load through a pointer to an int indicates that the load references an allocated int object. The pointer to an int may actually reference a pointer-aliased int symbol or an int member of a structure or union.

An effects-class signature is a subset of all the effects classes that might be referenced by a tuple. There is only one requirement for an effects-class signature: If two tuples may refer to the same part of memory, the intersection of their respective effects-class signatures must be non-null. If two tuples cannot refer to the same part of memory, it is desirable that the intersection of their effects-class signatures is null. An empty intersection leads to more optimization opportunities.

The most obvious rule for building an effects-class signature is to include in it all the effects classes that might be touched by a reference to the effects class. This leads to suboptimal code in cases such as that shown in Figure 4.

There are three effects classes for this code, s<0,3>, s<4,7>, and s<0,7>, generated by references to s.x, s.y, and s, respectively. If the effects-class signature for s<0,3> includes both s<0,3> and s<0,7> and the effects-class signature for s<4,7> includes both s<4,7> and s<0,7>, then the intersection of these two effects-

Table 2
Effects Classes Using the Standard C Type Rules

Effects Class	Type or Symbol	Start Offset	End Offset	Source Generating Effects Class
1	s	0	11	s
2	s	4	11	s.t
3	sp	0	7	sp
4	fp	0	7	fp
5	ip	0	7	ip
6	uip	0	7	uip
7	struct S	0	11	*sp
8	struct S	4	11	sp->t
9	struct S	4	7	sp->t.y
10	float	0	3	*fp
11	int	0	3	*uip and *ip

Table 3
Effects Classes Using the Traditional C Type Rules

Effects Class	Type or Symbol	Start Offset	End Offset	Source Generating Effects Class
1	s	0	11	s
2	s	4	11	s.t
3	sp	0	7	sp
4	fp	0	7	fp
5	ip	0	7	ip
6	uip	0	7	uip
7	char	0	1	*sp, sp->t, *uip, sp->t.y, *fp, *ip

class signatures is non-null. This falsely indicates that s.x and s.y may refer to the same memory location. This forces GEM to generate code that stores s.y after storing to s.x.

The DEC C and C++ side-effects package uses more effective rules for building effects-class signatures. These rules offer more optimization opportunities while preserving necessary dependency information.

Effects-class Signatures for Symbols If an effects class represents a region A of a symbol, its signature includes itself. Its signature also includes all effects classes representing regions of the symbol wholly contained within A. Finally, it includes any effects class representing a region of the symbol that partially overlaps A. It does not include effects classes representing regions of the symbol that do not overlap A or that wholly contain A.

Table 4 gives the symbol effects-class signatures for the three effects classes under discussion.

The inclusion of subregions in an effects-class signature means that references to symbols interfere with references to members therein and vice versa. Excluding super-regions in an effects-class signature means that references to two separate members of a symbol do not interfere with each other. In Table 4, the effects-class signatures for s<0,3> and s<4,7> do not interfere with each other. Both signatures interfere with the effects-class signature for s<0,7>.

The inclusion of effects classes representing partially overlapping regions of a symbol allows for the correct representation of the side effects of referencing submembers of complex unions.

```
struct S {
   int x;
   int y;
} s;
s.x = ...;
s.y = ...;
return s;
```

Figure 4
Example of Problematic Code for the Naïve Rule for Building Effects-class Signatures

Table 4
Symbol Effects-class Signatures

Effects Class	Effects-class Signature
s<0,3>	s<0,3>
s<4,7>	s<4,7>
s<0,7>	<0,3>, s<4,7>, s<0,7>

Effects-class Signatures for Types If an effects class represents a region of a type, the contents of its signature depends upon the type. If the type is the char type, the effects-class signature contains all the effects classes representing regions of other types or pointer-aliased symbols. This reflects the C and C++ type rules, which state that a pointer to a char can point to anything.

If the type is some type T other than char, the effects-class signature contains effects classes representing:

- Those regions of T that overlap the region of T the effects class represents, using the same overlap rules as for symbols
- Any region of a pointer-aliased symbol whose type is compatible to T, ignoring type qualifiers and signedness
- A region of a pointer-aliased aggregate or union symbol that contains a member or submember whose type is compatible to T, ignoring type qualifiers and signedness
- A region of an aggregate or union type that contains a member or submember whose type is compatible to T, ignoring type qualifiers and signedness

Table 5 gives the signatures for the effects classes in Table 2, assuming that the symbol s is pointer aliased.

Including the effects classes of symbols in the effects-class signatures of types records the interference of references through pointers with references to pointer-aliased symbols. In Figure 3, the pointer uip points to an unsigned int. The member s.t.y has type int. Thus, uip may point to s.t.y. The member s.t contains s.t.y. Thus, the signature for the effects-class int<0,3> con-

Table 5
Type Effects-class Signatures

Number	Effects Class	Effects-class Signature
1	s<0,11>	1, 2
2	s<4,11>	2
3	sp<0,7>	3
4	fp<0,7>	4
5	ip<0,7>	5
6	uip<0,7>	6
7	struct S<0,11>	1, 2, 7, 8, 9
8	struct S<4,11>	1, 2, 8, 9
9	struct S<4,7>	1, 2, 9
10	float<0,3>	1, 2, 7, 8, 10
11	int<0,3>	1, 2, 7, 8, 9, 11

tains the effects-class s<4,11>. This means that the load of s.t depends upon the store through uip.

Including the effects classes of types in the signatures of the effects classes of other types records the interference of references through a pointer with references through pointers to other types. In Figure 3, the pointer fp points to a float object. The member sp->t.z has type float. Thus, fp may point to sp->t.z. The member sp->t contains sp->t.z. Thus, the signature for the effects-class float<0,3> contains the effects-class struct S<4,11>. This reflects the fact that the store to sp->t.y depends upon the store through fp, i.e., it must occur after the store through fp.

Even though the signature for the effects-class float<0,3> contains the effects-class struct S<4,11> (sp->t), it does not contain the effects-class struct S<4,7> (sp->t.y). There is no float member of struct S whose position within struct S overlaps bytes 4 through 7 of struct S. There is a float member of struct S, namely z, whose position within struct S overlaps bytes 4 through 11 of struct S. The signature for the effects-class float<0,3> would not contain the effects-class s<0,3> if it existed. There is no float member of s whose position overlaps bytes 0 through 3 of s.

Additional Effects-class Signatures The side-effects package creates a special effects-class signature representing the side effects of a call. A called procedure may reference the following:

- Any pointer-aliased symbol (by means of a reference through a pointer)
- Any allocated object (by means of a reference through a pointer)
- Any nonlocal symbol (by means of direct access)
- Any local static symbol (by means of recursion)

The effects signature for a call includes all the effects classes representing these objects.

Responding to Optimizer Queries During optimization, the optimizer makes two types of queries to the side-effects analysis routines: dominator-based queries and nondominator-based queries.

When doing nondominator-based optimizations, the optimizer uses a bit vector to represent those objects a write may change (its effects). A similar bit vector represents those objects whose value a read may fetch (its dependencies). Each bit in the bit vector represents an effects class. If a tuple's effects-class signature contains an effects class, that effects class's bit is set in the tuple's bit vector. The optimizer uses the union of the bit vectors associated with a set of tuples to represent the combined effects or dependencies of those tuples.

Dominator-based queries involve finding the nearest dominating tuple that might write to the same memory location as the tuple in question. Tuple A dominates tuple B if every path from the start of the routine to B goes through A.[8] If both tuples A and C dominate B, tuple A is the nearer dominator if C dominates A.

When doing dominator-based optimizations, the side-effects package represents the tuples in the current dominator chain as a stack, adding and removing tuples from the stack as GEM moves from one path in the routine's dominator tree to another. Searching a single stack for the nearest dominating tuple that might write the same memory as the tuple in question references could lead to $O(N^2)$ performance, where N is the number of tuples in the dominator chain. This worst-case behavior occurs when none of the tuples in a dominator chain affects any subsequent tuple in the chain. Each time the side-effects package searches the stack, it examines all the tuples in the stack.

To avoid this, the DEC C and C++ side-effects package creates a stack for each effects class. When pushing a tuple, the side-effects package pushes the tuple on each stack associated with an effects class in the tuple's effects-class signature. When the GEM optimizer tells the side-effects package to find the nearest dominating write for a tuple, the side-effects package need only choose the nearest of those tuples that are on the top of the stacks associated with the tuple's effects-class signature. It need only look at the top of each stack, because a tuple would not be in the stack unless it might affect objects in the effects class associated with the stack.

The multistack worst-case behavior is $O(NC)$. There are C separate stacks, one for each effects class. The effects-class signature for each effects class may contain all the other effects classes. This would mean that each of the N tuples in the dominator chain would appear in each of the stacks.

Although the worst-case behavior for the multistack case is no better than the single-stack case (C may be equal to N), in practice there are often more tuples within a routine than effects classes. Furthermore,

effects-class signatures often contain a small number of effects classes. A small number of effects classes in an effects-class signature means that there are a small number of stacks to consider. Choosing the nearest dominator from among the top tuples on these stacks requires examining only a small number of tuples.

Cost of Using Type Information

When compiling all of the SPECint95 test suite[9] using high optimization, alias analysis accounts for approximately 5 percent of the compilation time. The use of Standard C type rules during alias analysis increases compilation time by less than 0.2 percent (time measured in number of cycles consumed by the compiler as reported by Digital Continuous Profiling Infrastructure [DCPI][10]). The increase in compilation time varies from program to program but never exceeds 0.5 percent. Handling the extra effects classes generated by using Standard C type aliasing information accounted for most of the increase.

Potentially, the cost of including type-aliasing information could be huge. Calculating which effects classes a reference through a char * pointer could touch is straightforward as shown by the algorithm in Figure 5.

A much more complicated process is required to calculate which effects classes could be touched by a reference through a pointer to a type other than char. The algorithm in Figure 6 performs this process.

Fortunately, the innermost section of this loop is rarely executed. The innermost section executes only if a routine references a structure either through a pointer or a pointer-aliased symbol, that structure contains a substructure, and the routine references the substructure through a pointer.

Effectiveness

The benchmark programs from the SPECint95 suite offer some convenient test cases for measuring the effectiveness of type-based alias analysis. The sources are readily available and portable. The programs conform to alias rules established by the American National Standards Institute (ANSI) and are compute intensive. Unfortunately, they do not contain floating-point calculations. This reduces the number of different types used in the programs. Type-based alias analysis works best when there are many different types in use.

Three of the SPECint95 programs show no improvement when compiled using the Standard C typing rules as opposed to using the traditional C typing rules. These programs, namely compress, go, and li, do not use many different types and pointers to them. When all the pointers in a program are pointers to ints (go), there is only one effects class for all pointer accesses. Because the compiler has no way to differentiate among the objects touched by a dereference of a pointer expression, it generates identical code for these programs, regardless of the type rules used. The generated code for li differs only slightly and only for infrequently executed routines.

Changes in generated code for the remaining five benchmarks are more prevalent. Two benchmarks, ijpeg and perl, show a small reduction in the number of loads executed but no meaningful reduction in the total number of instructions executed. The other three SPECint95 benchmarks show varying degrees of reduction in both the number of loads executed (see Table 6) and the total number of instructions executed (see Table 7).

```
foreach pointer aliased symbol
   foreach effects class representing a region of the symbol
       add that effects class to the effects class signature for char
```

Figure 5
Calculation of the Effects-class Signature of the Type char *

```
foreach pointer aliased symbol or type referenced through a pointer
   foreach member therein
      if the member's type is referenced through pointer
         foreach effects class representing a region of the member's type
            foreach effects class representing a region of the symbol or type
                        referenced through a pointer
               if the two effects class regions overlap
                  add the symbol's or pointer's effects class to the effects
                        class signature associated with the effect class
                        representing the member's type
```

Figure 6
Calculation of the Effects-class Signature for Types Other Than char

Table 6
Number of Loads Executed by the Select SPECint95 Benchmarks

SPEC Benchmark	Millions of Loads Using Type Information	Millions of Loads without Type Information	Percent Reduction
gcc	10,268	10,365	0.9
ijpeg	16,853	16,888	0.2
m88ksim	13,889	14,157	1.9
perl	11,260	11,296	0.3
vortex	18,994	19,207	1.1

Table 7
Number of Instructions Executed by the Select SPECint95 Benchmarks

SPEC Benchmark	Millions of Instructions Using Type Information	Millions of Instructions without Type Information	Percent Reduction
gcc	42,830	42,935	0.2
ijpeg	82,844	82,834	0.0
m88ksim	72,490	73,155	0.9
perl	45,219	45,252	0.1
vortex	80,093	80,607	0.6

The load and instruction counts are those reported by using Atom's pixie tool on the SPECint95 binaries to generate pixstat data.[11,12] The compiler used was a development C compiler. All compilations used the following switches: `-fast, -O4, -arch ev56,` and `-inline speed.` The compilations using the Standard C type system used the `-ansi_alias` switch. The compilations using the traditional C type system used the `-noansi_alias` switch. The benchmark binaries were run using the reference data set.

DCPI[10] measurements of the reduction in the number of cycles consumed by these SPECint95 benchmarks showed no consistent reductions. Run-to-run variability in the data collected swamped any cycle-time reductions that might have occurred. Similarly, measurements of gains in SPECint95[9] results due to the use of type information during alias analysis showed no significant changes.

Changes in Generated Code

The code-generation changes one sees in the SPECint95 benchmarks are exactly what one would expect.

The use of type information during alias analysis reduces the number of redundant loads. An example of this occurs in ijpeg, which contains the code sequence:

```
main->rowgroup_ctr
   = (JDIMENSION)(cinfo->min_DCT_scaled_size + 1);
main->rowgroups_avail
   = (JDIMENSION)(cinfo->min_DCT_scaled_size + 2);
```

in process_data_context. Using the traditional C type system, the compiler must assume that main->row group_ctr is an alias for cinfo->min_DCT_scaled_size. Thus, it must generate code that loads cinfo->min_ DCT_scaled_size twice. The Standard C type system allows the compiler to generate only one load of cinfo->min_DCT_scaled_size.

Several of the benchmarks contain code similar to the following from conversion_recipe in gcc:

```
curr.next->list->opcode = -1;
curr.next->list->to = from;
curr.next->list->cost = 0;
curr.next->list->prev = 0;
```

Using traditional C type rules, the compiler must generate four loads of curr.next->list. The compiler must assume that the pointer curr.next->list may point to itself, making curr.next->list->member an alias for curr.next->list. The Standard C type rules allow the compiler to assume that curr.next->list does not point to itself. This allows the compiler to generate code that reuses the result of the first load of curr.next->list, eliminating three redundant loads.

In another example in gcc, the use of Standard C type rules allows the compiler to move a load outside a loop. The following loop occurs in fixup_gotos:

```
for (; lists; lists = TREE_CHAIN (lists))
  if (TREE_CHAIN (lists)
        == thisblock->data.block.outer_cleanups)
     TREE_ADDRESSABLE (lists) = 1
```

Standard C type rules tell the compiler that the store generated by TREE_ADDRESSABLE (lists) = 1 cannot modify thisblock->data.block.outer_cleanups. This allows the compiler to generate code that fetches thisblock->data.block.outer_cleanups once before entering the loop. Using traditional C type rules, the compiler must generate code that fetches

thisblock->data.block.outer_cleanups each time it traverses the loop.

Not only can type information reduce the number of redundant loads, it can reduce the number of redundant stores. In m88ksim, there are many routines similar to the following:

```
int ffirst(struct instruction *cmd, union opcode *ptr) {
  ...
  ptr->gen.opc1 = 0x3d;
  ptr->gen.dest = operands.value[0];
  ptr->gen.opc2 = cmd->opc.rrr;
  ptr->gen.src2 = operands.value[1];
  return(0);
}
```

where opc1, dest, opc2, and src2 are bit fields sharing the same 32 bits (longword). Using traditional C typing rules, ptr->gen and cmd->opc may be aliases for each other. Thus to implement the above routine, the compiler must generate code that performs the following actions:

- Load ptr->gen
- Update bit fields ptr->gen.opc1 and ptr->gen.dest
- Store ptr->gen
- Load cmd->opc.rrr
- Update bit fields ptr->gen.opc2 and ptr->gen.src2
- Store ptr->gen

Using Standard C typing rules, the compiler does not have to generate the first store of ptr->gen. The assignments to ptr->gen.opc1 and ptr->gen.dest cannot change cmd->opc.rrr. In this case, alias analysis that is not type based would have a difficult time detecting that ptr->gen and cmd->opc do not alias each other. M88ksim never calls ffirst directly. It calls it by means of an array-indexed function pointer.

A Note of Caution

Many C programs do not adhere to the Standard C aliasing rules. Through the use of explicit casting and implicit casting, they access objects of one type by means of pointers to other types. More aggressive optimization by GEM combined with more detailed alias-analysis information from the DEC C and C++ side-effects package increasingly results in these programs exhibiting unexpected behavior when the compiler uses Standard C aliasing rules.

Passing a pointer to one type to a routine that expects a pointer to another type works as expected, until the GEM optimizer inlines the called procedure. If the procedure is not inlined, the DEC C and C++ side-effects package must assume that the call conflicts with all pointer accesses before and after the call. Once GEM inlines the routine, the side-effects package is free to assume that references using the inlined pointer do not conflict with references using the pointer at the call site. The two pointers point to two different types.

A recent example of this problem occurred in the gcc program in the SPECint95 benchmark suite. All programs in this suite are supposed to conform to the Standard C type-aliasing rules. Because of an improvement to the GEM optimizer, this benchmark started to give unexpected results. In rtx_alloc, gcc clears a structure by treating it as an array of ints, assigning zero to each element of the array. Subsequent to zeroing this structure, gcc assigns a value to one of the fields in the structure. Through a series of valid optimizations (given the incorrect type information), the resulting code did not clear all the fields in the structure. This left uninitialized data in the structure, resulting in gcc behaving in an unexpected manner.

To avoid potential problems, the DEC C compiler, by default, does not use the Standard C type rules when performing alias analysis. The user of the compiler has to explicitly assert that the program does follow the Standard C type rules through the use of a command-line switch.

The DIGITAL C++ compiler does assume that the C++ program it is compiling adheres to the Standard C++ type rules. A user of the DIGITAL C++ compiler can use a command-line switch to inform the compiler that it should use traditional C type rules when performing alias analysis.

Summary

Using Standard C type information during alias analysis does improve the generated code for some C and C++ programs. The compilation cost of using type information is small. Except for rare cases, performance gains resulting from these code improvements are small. Any programs compiled using type information during alias analysis must strictly adhere to the Standard C and C++ aliasing rules. If not, the optimizer may generate code that produces unexpected results.

Acknowledgments

The author would like to thank Dave Blickstein, Mark Davis, Neil Faiman, Steve Hobbs, and Bill Noyce of the GEM team for their advice and reviews of this work. Dave Blickstein and Neil Faiman also did work in the GEM optimizer to ensure that the DEC C and C++ side-effects package had all the information it needed to do alias analysis correctly and to ensure that the GEM optimizer effectively used the information the side-effects package provided. Thanks also to John Henning of the CSD Performance Group and Jeannie Lieb of the GEM team for their help using the SPECint95 benchmark suite. A final word of thanks goes to Bob Morgan for suggesting that I write this paper and to my management for supporting my doing so.

References and Notes

1. R. Wilson and M. Lam, "Efficient Context-Sensitive Pointer Analysis for C Programs," *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation,* La Jolla, Calif. (June 1995): 1–12.
2. D. Coutant, "Retargetable High-Level Alias Analysis," *Proceedings of the 13th Annual Symposium on Principles of Programming Languages,* St. Petersburg Beach, Fla. (January 1986): 110–118.
3. A. Diwan et al., "Type-Based Alias Analysis," *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation,* Montreal, Canada (June 1998): 106–117.
4. Joint Technical Committee ISO/IEC JTC 1, "The C Programming Language," *International Standard ISO/IEC 9899:1990,* section 6.3 Expressions.
5. "Working Paper for Draft Proposed International Standard for Information Systems—Programming Language C++," WG21/N1146, November 1997, section 3.10.
6. D. Blickstein et al., "The GEM Optimizing Compiler System," *Digital Technical Journal,* vol. 4, no. 4 (Special Issue, 1992): 121–136.
7. R. Crowell et al., "The GEM Loop Transformer," *Digital Technical Journal,* vol. 10, no. 2, accepted for publication.
8. A. Aho, R. Sethi, and J. Ullman, *Compilers Principles, Techniques, and Tools* (Reading, Mass: Addison-Wesley, 1986): 104.
9. Information about the SPEC benchmarks is available from the Standard Performance Evaluation Corporation at http://www.specbench.org/.
10. J. Anderson et al., "Continuous Profiling: Where Have All the Cycles Gone?" *Proceedings of the Sixteenth ACM Symposium on Operating System Principles,* Saint-Malo, France (October 1997): 15–26.
11. A. Srivastava and A. Eustace, "ATOM: A System for Building Customized Program Analysis Tools," *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation,* Orlando, Fla. (June 1994): 196–205.
12. *UMIPS-V Reference Manual (pixie and pixstats)* (Sunnyvale, Calif.: MIPS Computer Systems, 1990).

Biography

August G. Reinig
August Reinig is a principal software engineer, currently working on debugger support in the DIGITAL C++ compiler. In addition to his work on the DEC C and C++ side-effects package, August implemented a Java-based distributed test system for the DEC C and DIGITAL C++ compilers and a parallel build system for the DEC C and DIGITAL C++ compilers. The distributed test system simultaneously runs multiple tests on different machines and is fault tolerant. Before joining the DEC C and C++ team, he contributed to an advanced development incremental compiler project, which led to two patents, "Method and Apparatus for Software Testing Using a Testing Technique to Test Compilers" and "Method and Apparatus for Testing Software." He earned a B.S. in mathematics (magna cum laude) from Dartmouth College in 1980 and an M.S. in computer science from Harvard University in 1997. He is a member of Phi Beta Kappa.