Alias Analysis for Intermediate Code

Sanjiv K. Gupta Naveen Sharma
System Software Group
HCL Technologies, Noida, India – 201 301
{sanjivg,naveens}@noida.hcltech.com

Abstract

Most existing alias analysis techniques are formulated in terms of high-level language constructs and are unable to cope with pointer arithmetic. For machines that do not have 'base + offset' addressing mode, pointer arithmetic is necessary to compute a pointer to the desired address. Most state of the art compilers such as GCC lack the mechanism to determine aliasing between such computed pointers. Few other existing alias analysis techniques described for executable code can handle pointer arithmetic but require large memory when applied to intermediate languages such as RTL. In this paper, we describe a method of disambiguating the computed pointers within a procedure at the intermediate code level. The method is similar to the techniques described for executable code but requires significantly less amount of memory. We have experimented our method with the GCC RTL and it reduces the code size of array manipulating benchmarks by approximately 4-7% for the machines that do not have 'base + offset' addressing mode.

1 Introduction

Various optimization passes like CSE and instruction scheduling rely on alias analysis to determine the aliasing between two memory references. Compile time alias analysis in compilers such as GCC can successfully determine aliasing between two memory references if they (i) use distinct offsets from the same register; or (ii) one of them points to stack. But such compile time alias analysis often fails to determine aliasing between computed pointers and safely assume that these pointers may alias.

To illustrate the computed pointers and aliasing problem with them, let us consider the following piece of code:

```
void foo (double *in)
{
    in[4] += in[3];
}
```

For machines that have 'base + offset' addressing for `double`, GCC generates RTL like,

```
r172 = [r170, 32]
r173 = [r170, 24]
r174 = r172 + r173
[r170, 32] = r174
```

in such cases the GCC can successfully determine that the memory references `[r170, 32]` and `[r170, 24]` do not alias as they use distinct offsets from the same base register.

On machines that do not have 'base + offset' addressing mode for `double`, the compiler will need to compute the pointers to load and store locations. In these cases, the generated RTL will look like,

```
r170 = r160 + 32
r171 = r160 + 24
r172 = [r170]
r173 = [r171]
r174 = r172 + r173
[r170] = r174
```

GCC fails to determine aliasing between the computed pointers `r170` and `r171`. To be safe, GCC simply assumes that these computed pointers alias with each other. The problem with GCC is that it does not have any mechanism to keep track of what address arithmetic have been performed to obtain the computed pointers `r170` and `r171`.

There are algorithms available to keep track of address arithmetic (see [Debray98]); but they work well only with the executable code since the executable programs have small number of registers (i.e. only hard registers). Time and space requirements of such algorithms increase when we try to use them for intermediate code such as RTL as there may be large number of pseudo registers present in the intermediate code. This paper describes an alias analysis algorithm that can be used with the intermediate code to keep track of the address arithmetic efficiently. The algorithm is influenced from the mod-k residue technique for executable code described in [Debray98].

2 Terminology

The mod-k residue algorithm maps each pseudo with a set of possible address values at each program point. Let us first define some basic terms that are required to discuss the algorithm. The term pseudo means a pseudo register in entire discussion.

2.1 A Program Point

A program point refers to a point between two instructions[Muchnick]. A program point *p* between instructions *I1* and *I2* is denoted as *p(I1, I2)*, where *I1* immediately preceds *p* and *I2* immediately follows *p*. Since compilers always keep a chain of instructions available all the time, the preceding instruction *I1* is all which is required to identify a program point *p*.

Any solution that attempts to disambiguate two computed pointers should be able to tell the possible address values represented by each pointer pseudo at each program point. For example, for the pointer pseudos `r1` and `r2` at given program points *p1* and *p2*, the solution must be able to tell the possible address values represented by `r1` at *p1*, and `r2` at *p2*.

2.2 mod-k Residues Set

For compactly storing an address value we consider only some fixed number, say *m*, of the lower bits of the value. This means an abstract address value *val* is represented by its mod-k residue *val mod k*. $(k = 2^m)$. The set of all abstract address values can then be represented by the mod-k residues set $Z = (0, 1, 2, ..., k-1)$. Since $(x \ mod \ k) \neq ((x + \delta) \ mod \ k) \ \forall \ 0 < \delta < k$, the representation can distinguish between addresses involving distinct "small" displacements δ (i.e. less than *k*) from a base register.

The choice of the value *k* is critical for efficiency of the technique. The value *k* determines the size of mod-k residues set; the choice should be made in such a way that it makes storing and manipulating mod-k residues sets low cost operations. Often, the natural word size of the host machine is a good choice. This way we can store a mod-k residue set as a bit vector in a single machine word. Operations such as adding a constant *c* to each member of the set can be simply obtained by rotating the bit-vector by *c* bits. For example, the mod-k residues set (4, 12) can be represented by a machine word whose 4^{th} and 12^{th} bits are ON and rest of the bits are OFF.

In our implementation experiment with GCC on host machine x86, we choose the value of *k* as sizeof(int) so that a mod-k residues set can be stored efficiently in an integer.

2.3 Address Descriptors

The mod-k residues sets by themselves are not adequate for cases where we are not able to predict the actual value of a pseudo `r` at a program point. To deal with this problem we extend mod-k residues set to 'Address Descriptors'. An address descriptor is a pair *{I, Z}*, where *I* is an instruction and *Z* is a mod-k residues set. Given an address descriptor *A*(`r`) = *{I, Z}* for a pseudo `r`, the instruction *I* is the defining instruction of `r`, and *Z* denotes the set of mod-k residues relative to whatever value is computed by instruction *I*.

The address descriptor of a pseudo `r` is computed by analyzing its defining instruction as per the rules described in section 3. If we cannot say anything about the value of a pseudo `r` while analyzing its defining instruction *I*, we associate the address descriptor *{I, (0)}* with `r`. A constant *c* yields an address descriptor *{NONE, (c mod k)}*.

For example consider the following instructions:

```
I1: r172 = [r170]
I2: r173 = 5
I3:
```

The address descriptor of pseudo `r172` at program point *p(I1, I2)* will be *{I1, (0)}* as we can not say anything about the value of `r172` after instruction *I1*. The address descriptor for pseudo `r173` at program point *p(I2, I3)* will be *{NONE, (5)}*.

Further we define two special address descriptors, an address descriptor *{ANY, (all)}* alias with everything and the address descriptor *{NONE, (nothing)}* alias with nothing.

3 Effect of Individual Instructions on Address Descriptors: Keeping track of Address Arithmetic

The operations performed by an instruction modifies certain pseudos; the algorithm defines these operations for address descriptors and applies them to modify address descriptors corresponding to those pseudos. In this section we define assignment, addition, and multiplication operations for address descriptors as they are the most frequent operations occurring in address arithmetic.

3.1 Assignment Instructions

Consider an assignment instruction *I* having the following form,

```
I: dest = src
```

where `dest` is a pseudo and `src` could be a pseudo or some integer constant.

The address descriptor of `dest` pseudo is evaluated as following:

a) If `src` is a pseudo and has a valid address descriptor, the address descriptor of `src` becomes the address descriptor of `dest`.

b) If `src` is a pseudo that does not have a valid address descriptor, the address descriptor of `dest` becomes *{I,(0)}*.

c) If `src` is a constant integer *c*, address descriptor of `dest` will be as *{NONE, (c mod k)}*.

3.2 Addition Instructions

Consider an addition instruction *I* having the following form.

```
I: dest = src1 + src2
```

where `dest` and `src1` are pseudos and `src2` can be a pseudo or an integer constant. Let *{I1, Z1}* and *{I2, Z2}* be the address descriptors of `src1` and `src2` respectively. The address descriptor of pseudo `dest` is then evaluated as following:

a) If *I1 = NONE*, the address descriptor of `dest` becomes *{I2, Z}* (the situation where *I2 = NONE* is symmetric). Here $Z = \{((x + y)\ mod\ k)\ \forall\ x \in Z1, y \in Z2\}$.

b) Otherwise, we can not say anything about the result of this operation. So the address descriptor of `dest` after this instruction *I* is taken to be *{I, (0)}*.

3.3 Multiplication Instructions

Consider a multiplication instruction I having the following form,

```
I: dest = src1 * src2
```

where `dest` and `src1` are pseudos and `src2` can be a pseudo or an integer constant. Let *{I1, Z1}* and *{I2, Z2}* be the address descriptors of `src1` and `src2` respectively. The address descriptor of `dest` is then evaluated as following:

a) If *I1 = NONE*, the address descriptor of `dest` becomes *{I2, Z}* (the situation where *I2 = NONE* is symmetric). Here $Z = \{((x * y)\ mod\ k)\ \forall\ x \in Z1, y \in Z2\}$.

b) Otherwise, we can not say anything about the result of this operation. So the address descriptor of `dest` after this instruction *I* is taken to be *{I, (0)}*.

Though semantics for other operations on address descriptors can be defined but above integral operations suffice in most cases to handle the pointer arithmetic. The address descriptor of the destination pseudo `r` of an unhandled-instruction[1] *I* is taken as *{I,(0)}*.

[1]instruction for which the corresponding address descriptor operation is not defined

4 The Algorithm

The algorithm maps each pseudo with its possible values (i.e. an address descriptor) at each program point. Since storing an address descriptor for each pseudo at each program point will require excessive memory, we compute the address descriptors of pseudos defined in a basic block and store them only at the end of the basic block. Using this saved information, the address descriptor of a pseudo at a particular program point within a basic block can be obtained by recomputing the address descriptors of the basic block upto that program point. This recomputing does not take much time as basic blocks happen to be small in most cases.

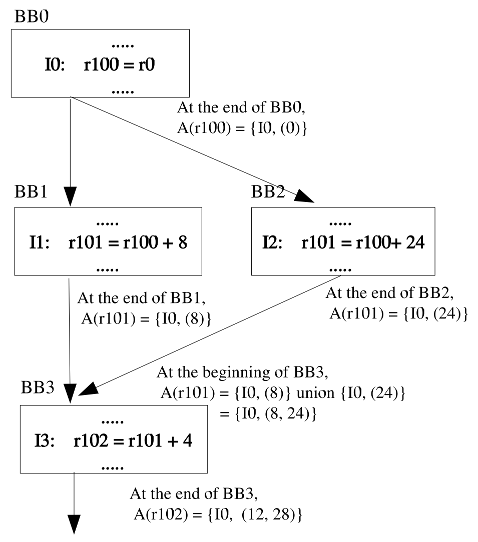
4.1 Computing Address Descriptors

The instructions in a basic block are analyzed as described in section 3 to compute the address descriptors of pseudos defined in that basic block. The input address descriptors of the basic block are determined as described in the subsection 4.2. The address descriptors computed in the basic block are then saved at the end of that basic block. This saved list of address descriptors at the end of a basic block is called *OUT_LIST* of that basic block.

Storing the address descriptors for all pseudos defined in a basic block in the *OUT_LIST* will require very large memory (intermediate code may contain large number of pseudos). Since most of the defined pseudos are local to a basic block, they do not contribute to the input of their successors. To reduce the memory requirements, address descriptors for such pseudos need not be saved in the *OUT_LIST*. The algorithm first identifies all those pseudos that are being used across basic blocks. We call such pseudos as “shared pseudos”. The address descriptors for “shared pseudos” only are saved at the end of all basic blocks. This saves lot of memory since there exists usually a small number of “shared pseudos” in intermediate code. If a procedure with *N* basic blocks have *R* shared pseudos, the memory required for storing the address descriptors would be *RN(k+w)* bits, where *w* is the machine word size in bits.

4.2 Propagating Address Descriptors across Basic Blocks

CFG is used to propagate these descriptors across basic blocks. A *union* operation is used to “merge” the information coming along the incoming edges at vertices



Notation: A(r) denotes the address descriptor of register 'r'

Figure 1: Merging of address descriptors

in the CFG. An input list of address descriptors (we call this an *IN_LIST*) for a basic block is formed by doing the *union* of OUT_LISTs of its predecessors. Thus if the address descriptors for a pseudo `r` being propagated along two incoming edges at a vertex in the CFG are *{I1, Z1}* and *{I2, Z2}*, the resulting address descriptor for pseudo `r` is obtained as,

{I, Z1 union Z2} if *I1=I2=I*.
{ANY, (all)} if $I1 \neq I2$.

For example, as shown in Figure 1, consider a basic block `BB3` having two predecessors `BB1` and `BB2`. If the address descriptors of a pseudo `r101` in the *OUT_LISTs* of `BB1` and `BB2` are *{I0, (8)}* and *{I0, (24)}*, the address descriptor of `r101` in the *IN_LIST* of `BB3` will be *{I0, (8, 24)}*.

4.3 Building the Fixed alias analysis Information

Multiple iterations over the CFG are done till the address descriptors of all “shared registers” in a procedure become constant, or in other words till the *OUT_LISTs* of all basic blocks become constant. Each iteration computes the *OUT_LIST* of each basic block using the

IN_LIST of the basic block as input. The *OUT_LIST* computed during the iteration is *union*ed (as described in subsection 4.2) with the saved *OUT_LIST* of the previous iteration and the result is saved as the current *OUT_LIST* of the basic block. Another iteration over CFG is required only if any of the *OUT_LISTs* change in the current iteration. The required information for performing alias analysis is built once we have reached this stage where all the *OUT_LISTs* are fixed. This way we have gathered for all "shared pseudos", all the possible results of operations performed on them by all execution paths.

We can describe this in the following pseudocode,

```
do {
  out_lists_changed = false;
  for each BB in the CFG {
    prepare an IN_LIST of BB
      by doing union of the
      OUT_LISTS of
      predecessors of BB;
    evaluate OUT_LIST_OF_THIS_PASS
      using IN_LIST as input;
    NEW_OUT_LIST = do union of
      OUT_LIST_OF_THIS_PASS
      with the SAVED_OUT_LIST.
    list_changed = false;
    if (NEW_OUT_LIST is not
      equal to SAVED_OUT_LIST) {
      SAVED_OUT_LIST = NEW_OUT_LIST;
      list_changed = true;
    }
    out_lists_changed =
     out_lists_changed | list_changed;
  }
} while (out_lists_changed);
```

To reduce the number of iterations required over CFG, we identify loop counters such as `r = r + const` and populate their address descriptor in a single pass itself. For example, given a loop counter `r` in RTL below,

```
I1: r = 0
...
I7: r = r + 2
```

The address descriptor of pseudo `r` is calculated in the first pass itself as *{NONE, (0,2,4,6,8,10,12,14)}* (for mod-16 alias analysis).

5 Reasoning about alias relationship

Once the required alias information is generated, the aliasing relationship between two computed pointers can be determined in following steps.

Step 1. Given two computed pointers `r1` and `r2`, we retrieve the program points *p1* and *p2* where `r1` and `r2` are dereferenced.

To retrieve the program points for these pointers a hash table is built at the start of the algorithm. For every pointer, this hash table records the instruction in which the pointer is contained. For a pointer, the instruction retreived from the hash table gives the preceding instruction of the program point.

Step 2. Find the basic blocks for the program points *p1* and *p2*; say they are `BB1` and `BB2`.

Step 3. Compute the *IN_LISTs* of `BB1` and `BB2` by doing the union of saved *OUT_LISTs* of their predecessors.

Step 4. Recompute the address descriptors *{I1, (Z1)}* and *{I2, (Z2)}* of the two pointers `r1` and `r2` at the desired program points *p1* and *p2* by traversing within their basic blocks `BB1` and `BB2`.

Step 5. Address descriptors *{I1, (Z1)}* at instruction point *p1*, and *{I2, (Z2)}* at instruction point *p2* denote disjoint addresses if both the following conditions are satisfied.

i) *I1 = I2 = 'I'*.

ii) *Z1 intersection Z2 = NULL*

Condition (i) ensures that both the program points *p1* and *p2* see the same value computed by instruction *I*. Condition (ii) then ensures that relative to this value, the pointer `r1` referred at *p1* is disjoint from the pointer `r2` referred at *p2*.

6 Example

Let us describe the entire algorithm with the help of the following example function in C,

```
void foo (double * a)
{
  int i, j;

  i = 0;
  j = 2;

  if (! a)
```

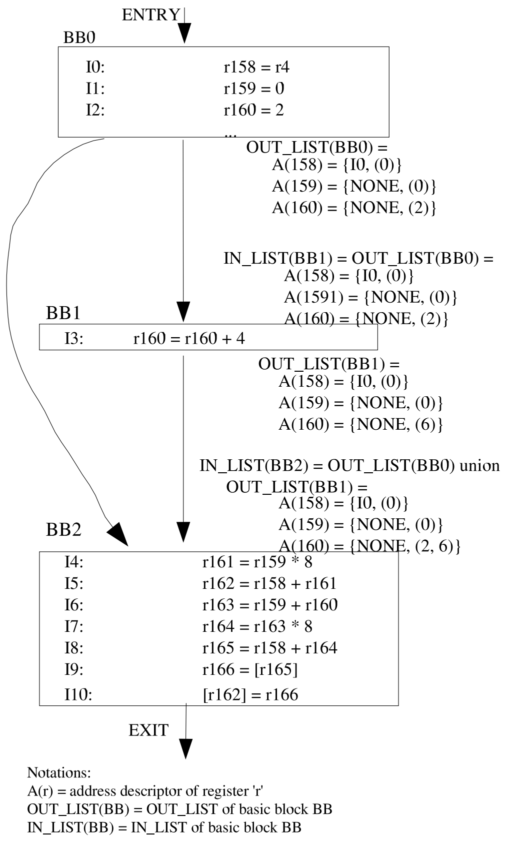


Figure 2: address descriptor based alias analysis

```
    j = j + 4;

  a[i] = a[i + j];
}
```

Figure 2 shows the CFG and RTL generated for SH4 alongwith the address descriptors computed by the algorithm. To determine the aliasing relationship between the computed pointers `r165` and `r162` in basic block BB2, their address descriptors are recomputed using the *IN_LIST* of BB2. Applying the rules of Section 3 on BB2 gives the recomputed address descriptors of `r162` and `r165` as *{I1, (0)}* and *{I1, (16)}*. These address descriptors do not alias since they follow the rules described in step5 of Section 5.

7 Drawbacks

The algorithm is not capable of keeping track of contents of memory. Information about a register is lost if it is saved to memory and then subsequently restored at a later point. Also if a register can have different defining instructions at different predecessors of a CFG vertex, the information is lost while merging them using the *union* operator.

The precision of results obtained also depends on the value of *k*. The algorithm can only distinguish between the displacements in the range $\{0 < \delta < k\}$. For example, if *k*=32 then the algorithm will not be able to differentiate between the computed pointers for `&in[9]` and `&in[13]`. Increasing the value of *k* improves the precision of results obtained but may also increase the execution time of algorithm.

8 Experimentation and Results

We experimented by implementing this algorithm in GCC. Since the compiler was running on an i686 machine, we chose the value of k as 32. We built the cross compiler for ia64-elf target and obtained the data about generated code size. Table 1 given below compares the generated code size for ia64-elf for some of stress-1.17 files with -O2 option. We also observed that our implementation increases the compilation time for programs by about 20%.

file name	**size of .text section (before)**	**size of .text section (after**	**%code size decrease**
dct64.o	9808	9568	2.44
lpc.o	36824	33256	9.68
mdct.o	5936	5488	7.54
polyobj.o	14840	14360	3.23
layer3.o	54760	54344	0.75
tif_lzw.o	24320	24256	0.32
quadrics.o	22000	21840	0.73

Table 1: code size comparison for ia64-elf

9 Acknowledgments

We would like to thank people at gcc@gcc.gnu.org for their invaluable support. We specially thank to Richard Henderson, Diego Novillo, Saumya K. Debray, Daniel Berlin and David Edelsohn for their ideas.

References

[Debray98] S. Debray, R. Muth, and M. Weippert, *Alias Analysis of Executable Code*, In The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 12-24, Orlando, Florida (1998)

[Muchnick] Steven S. Muchnick, *Advanced Compiler Design and Implementation*, Morgan Kaufmann Inc., Reading, page 303, USA (1997)

[GCC] GCC source code, `http://gcc.gnu.org`