Alias Analysis of Executable Code *

Saumya Debray Robert Muth Matthew Weippert
Department of Computer Science
University of Arizona
Tucson, AZ 85721, U.S.A.
{debray, muth, weippert}@cs.arizona.edu

Technical Report 97-13

July 1997

Abstract

Recent years have seen increasing interest in systems that reason about and manipulate executable code. Such systems can generally benefit from information about aliasing. Unfortunately, most existing alias analyses are formulated in terms of high-level language features, and are unable to cope with features, such as pointer arithmetic, that pervade executable programs. This paper describes a simple algorithm that can be used to obtain aliasing information for executable code. In order to be practical, the algorithm is careful to keep its memory requirements low, sacrificing precision where necessary to achieve this goal. Experimental results indicate that it is nevertheless able to provide a reasonable amount of information about memory references across a variety of benchmark programs.

*This work was supported in part by the National Science Foundation under grant CCR-9502826

1 Introduction

Recent years have seen increasing interest in reasoning about and manipulating executable files [5, 14, 19, 24, 26, 29, 30, 32]. When working with an executable file, we typically have information about the entire program—including, potentially, library functions—that is usually not available at compile time. Because of this, code manipulation and optimization at this level offers benefits that are difficult or impossible to obtain using traditional compilers. As with the compilation of source-level programs, code transformations on executable code can benefit greatly from pointer alias information. For example, to obtain the full benefits of a superscalar architecture such as the DEC Alpha, link-time optimizers such as Spike [5], alto [9], and OM [29] need to carry out instruction scheduling again after link-time optimizations. Without pointer alias information, however, the scheduler must be conservative in its treatment of all loads and stores, and this severely limits the amount of code reordering that is possible. As another example, it may be possible to scavenge registers at link-time, e.g., by examining the register usage of library functions, but the ability to use such scavenged registers effectively is likely to be limited in the absence of pointer alias information.

There is an extensive body of work on pointer alias analysis of various kinds (see Section 5). In almost all cases, these are high level analyses, carried out on representations of source programs in terms of source language constructs, and typically disregarding "nasty" features such as type casts, pointer arithmetic, and out-of-bounds array accesses. Such analyses turn out, unfortunately, to be of limited utility at the machine code level, because at this level all we have are the "nasty" features. The contents of registers and memory words are untyped bit-strings, so the issue of type casts is in some sense moot: everything is potentially an address. Memory accesses typically involve some address arithmetic to compute a base address into a register, followed by the use of a displacement off the base address to carry out the actual memory reference. Address arithmetic may also arise due to particular language features, e.g., the use of "tag bits" in dynamically typed languages to indicate the type of the value pointed at. Dereferencing operations in the executable code for such programs will involve nontrivial arithmetic involving the tag bits that is invisible—and irrelevant—at the source level (at the level of executable programs, we can't tell what source language a particular piece of code was derived from, and different components of a program might have been written in different source languages, so we must be able to deal with all such address arithmetic in a reasonable way). If the number of arguments to a function is large enough, some of the arguments may have to be passed on the stack. In such a case, the arguments passed on the stack will typically reside at the top of the caller's stack frame, and the callee will "reach into" the caller's frame to access them: this is nothing but an out-of-bounds array reference. Finally, executable programs may include library functions, in hand-written assembly code, that violate familiar and comfortable source-level assumptions, e.g., that execution does not jump out of the middle of one function and into the middle of another (this happens, for example, in some Fortran library routines). To illustrate some of the problems that arise, consider the fragment of C code shown in Figure 1, together with the corresponding assembly code.[1] The point to note is the extensive use of address arithmetic to access memory, even in this very simple program fragment. For example, in order to determine whether instructions (3) and (4) might write to the same memory location, we need to be able to reason about the contents of registers `r16` and `r17`, which are defined primarily through arithmetic operations. As this example illustrates, pointer arithmetic cannot be ignored during alias analysis at the machine code level.

In this paper, we describe a low-level, flow-sensitive, context-insensitive interprocedural pointer alias analysis algorithm, designed and implemented in the context of the `alto` link time optimizer [9], that can handle significant pointer arithmetic and features, such as out-of-bound references, that are ignored by most existing alias analysis algorithms.

For simplicity in the discussion that follows, we assume a more or less canonical RISC instruction set. Memory is accessed only through explicit load and store instructions, which have the form `load` reg_a, k`(`reg_b`)` and `store` reg_a, k`(`reg_b`)`, where k is a constant, and have the effect of reading from, or writing to, the location whose address is $k + contents\_of(reg_b)$. To model arithmetic we assume the instructions `add` src_1, src_2, $dest$ and `mult` src_1,

[1] The assembly code shown corresponds to that obtained using `gcc -O` on a DEC Alpha workstation, with some edits to enhance readability. On the Alpha, arguments to functions are typically passed in registers 16 …21, and register 30 is used as the stack pointer.

```
     Source Code                  Executable Code
int g(int *x, int *y)                                # arg1 in r16, arg2 in r17
{                            add r30, -32, r30      # allocate stack frame          (1)
                             store r26, 0(r30)      # save return address           (2)
   *x = 1;                   store 1, 0(r16)                                        (3)
   *y = 0;                   store 0, 0(r17)                                        (4)
   ...
}

int f(int x, int y)                                  # arg1 in r16, arg2 in r17
{                            add r30, -48, r30      # allocate stack frame          (6)
                             store r26, 0(r30)      # save return address           (7)
                             store r16, 20(r30)     # save r16 in x's stack slot    (8)
                             store r17, 16(r30)     # save r17 in y's stack slot    (9)
   ...                          ...
   g(&y, &x);                add r30, 16, r16       # r16 := &y                     (10)
                             add r30, 20, r17       # r17 := &x                     (11)
                             bsr r26, g             # r26 := return addr; goto g    (12)
   ...                          ...
}
```

Figure 1: A fragment of a C program and the corresponding assembly code

src_2, $dest$, where *dest* is a destination register and src_1 and src_2 are source registers; to simplify the discussion we abuse notation and allow either src_1 or src_2 to be an integer constant, denoting an immediate operand. These instructions compute, respectively, the sum and product of src_1 and src_2 into *dest* (many other operations can be expressed in terms of these, e.g., subtraction and register-to-register moves can be modelled in terms of addition: we do not consider these separately). In addition to these we assume the usual complement of tests, conditional jumps, and direct and indirect unconditional jumps: the only effect of these instructions is to determine the control flow graph of the program, so we do not consider them explicitly in the context of alias analysis. We also ignore operations on floating point registers, since it seems unlikely that such operations would be used for address computations.

2 Local Alias Analysis

For reasoning about memory references within a basic block, we can use a technique called *instruction inspection*, commonly used in compile-time instruction schedulers, where two memory references are taken to be non-conflicting if either (i) they use distinct offsets from the same base register; or (ii) one uses a register known to point to the stack and the other uses a register known to point to the global data area. This can be generalized to determine whether two address expressions e_1 and e_2 within the same block might refer to the same location, based on the following simple observation:

Proposition 2.1 *Suppose that a basic block B contains sequences of operations (equivalent to): ,,*

I_1 : `add` r_1, c_1, r_2; I_2 : `add` r_2, c_2, r_3; ..., I_k : `add` r_k, c_k, r *and*

I'_1 : `add` r'_1, c'_1, r'_2; I'_2 : `add` r_2, c'_2, r'_3; ..., I'_m : `add` r'_m, c'_m, r'

where $k, m \geq 0$, such that (i) I_j uses the definition of r_j in I_{j-1}, and I'_j uses the definition of r'_j in I'_{j-1}; (ii) either both I_0 and I'_0 use the same definition of r_0 in the block B, or neither use any definition of r_0 in B; and (iii) $\sum_{i=0}^{k} c_i \neq \sum_{i=0}^{k} c'_i$. Then, the value of register r immediately after instruction I_k is different from that of register r' immediately *after instruction I'_k.*

Unfortunately, this simple approach does not work if information about address arithmetic needs to be propagated across basic block boundaries. In the next section we describe a global analysis that can be used to handle this.

3 Global Alias Analysis: Mod-k Residues

3.1 The Basic Idea

An alias analysis will in general associate each register with a set of possible addresses at each program point, so we need to abstract sets of addresses to descriptions, or "abstract address sets." These need to be easy to compute and compactly representable, with operations such as union, intersection, checking containment, etc., that are cheap enough to be practical for the analysis of large programs. A simple way to satisfy these criteria is to consider only some fixed number—say, m—of the low order bits of an address. That is, addresses are represented by their mod-k residues, where $k = 2^m$. The set of all mod-k residues is $\mathbf{Z}_k = \{0, \ldots, k-1\}$. An abstract address set can then be represented as a bit vector of length k; since m—and, therefore, $k = 2^m$—is fixed, set operations such as union, intersection, checking containment, etc., can be carried out in $O(1)$ bit-vector operations. This representation can cope with address arithmetic, e.g., as illustrated in Figure 1, since such arithmetic translates in a straightforward way to mod-k arithmetic (see, for example, [16]). Finally, since x mod $k \neq (x \pm \delta)$ mod k for $0 < \delta < 2^m$, the representation can distinguish between addresses involving distinct "small" displacements (i.e., less than 2^m) from a base register.

It turns out that mod-k residues are not, by themselves, adequate for our purposes. The problem is that in many cases we won't be able to predict the actual value of a register `r` (e.g., the stack pointer) at a program point, which means we won't be able to say anything about a displacement k from `r`, i.e., the address corresponding to k`(r)`, either. To deal with this problem we extend abstract address sets to *address descriptors*, which take an additional component that refers to an instruction:

Definition 3.1 An *address descriptor* is a pair $\langle I, M\rangle$, where I is either an instruction or one of the distinguished values $\{\text{NONE}, \text{ANY}\}$, and M is a set of mod-k residues. Given an address descriptor $A \equiv \langle I, M\rangle$, the instruction I is said to be the *defining instruction* of A, while M is called the residue set of A. ■

The intuition is that given an address descriptor $\langle I, M\rangle$, M denotes a set of mod-k residues *relative to* whatever value is computed by instruction I. A value of NONE indicates that the corresponding residue set represents mod-k residues of absolute addresses, while a value of ANY indicates that the address descriptor denotes all possible addresses. More formally, suppose that we are given an operational semantics for the instruction set under consideration (such a semantics is conceptually simple, if somewhat tedious, to specify for the simple instruction set considered here: we omit a formal specification due to space constraints, and rely instead on the informal description of the instructions given at the end of Section 1). Given a program P and an instruction I in P, let $val_P(I)$ denote the set of values w such that, for some input to P, there is an execution path from the entry point of P to the instruction I that causes I to compute w into its destination register ($val_P(I) = \emptyset$ if I does not compute a value into a register, or if control never reaches I). Extend this to the special values NONE and ANY as follows: for any program P, $val_P(\text{NONE}) = \{0\}$, and $val_P(\text{ANY})$ is the set of all values. Then, for an analysis using mod-k residues, the set of addresses denoted by an address descriptor $A \equiv \langle I, X\rangle$ in P—that is, the "concretization" of A in the context of P—is:

$$conc_P(\langle I, X\rangle) = \{w + ik + x \mid w \in val_P(I), x \in X, i \in \mathbf{Z}^+\}.$$

The relative precision of different address descriptors can be characterized via the binary relation $\trianglelefteq$:

Definition 3.2 An address descriptor $\langle I_2, X_2 \rangle$ is *more precise* than a descriptor $\langle I_1, X_1 \rangle$, written $\langle I_1, X_1 \rangle \trianglelefteq \langle I_2, X_2 \rangle$, if and only if (i) $I_1 = \text{ANY}$ or $X_1 = \mathbf{Z}_k$; or (ii) $X_2 = \emptyset$; or (iii) $I_1 = I_2$ and $X_2 \subseteq X_1$. ■

It is straightforward to show that $\trianglelefteq$ is reflexive and transitive, i.e., a preorder. It can be extended to a partial order in the usual way: define the relation $\simeq$ as $A_1 \simeq A_2$ if and only if $A_1 \trianglelefteq A_2$ and $A_2 \trianglelefteq A_1$—it is easy to show that this is an equivalence relation—and consider the quotient of $\trianglelefteq$ with respect to $\simeq$. The set of address descriptors forms a lattice with respect to this partial order. In the remainder of this discussion, we abuse notation and write $\trianglelefteq$ to refer to the resulting partial order. In particular, the equivalence class containing $\langle I, \mathbf{Z}_k \rangle$ for all I, as well as $\langle \text{ANY}, M \rangle$ for all M, denotes a total lack of information, and is written as $\perp$; the equivalence class containing $\langle I, \emptyset \rangle$ for all I, denotes the empty set of addresses and is written as $\top$. Our analysis associates an address descriptor with each register at each program point of interest.[2] If a register `r` has an associated address descriptor $\langle I, M \rangle$ at a program point, we will sometimes abuse terminology and refer to instruction I as the defining instruction for `r` at that point.

Example 3.1 Suppose that we use mod-32 residues, and consider the following pair of instructions from Figure 1:

```
store 1, 0(r16)          (3)
store 0, 0(r17)          (4)
```

Assuming that the only call site for `g()` is that in `f()`, we can use instruction (6) as a defining instruction for `r30`, and thence for `r16` and `r17`—how this is done follows from the way individual instructions are handled, as discussed in Section 3.2.1—the address descriptors corresponding to the address expressions occurring in instructions (3) and (4) are as follows:

Instruction	*Address Expression*	*Address Descriptor*	
(3)	`0(r16)`	$\langle (6), \{16\} \rangle$	(from instruction (10))
(4)	`0(r17)`	$\langle (6), \{20\} \rangle$	(from instruction (11))

Using these address descriptors, we can reason as discussed in Section 3.3 and conclude that instructions (3) and (4) write to distinct memory locations. □

3.2 The Analysis Algorithm

3.2.1 Effects of Individual Instructions

As mentioned earlier, the defining instruction component of an address descriptor allows us to refer to mod-k residues relative to "whatever value is computed by the defining instruction." When examining an instruction I with destination register `r`, if we can't say anything about the value of `r` after instruction I, then instead of setting the address descriptor for `r` to $\perp$, we use I as the defining instruction for `r` and associate the address descriptor $\langle I, \{0\} \rangle$ with `r` at the point immediately after I. To simplify the discussion, we assume that an immediate operand c yields an address descriptor $\langle \text{NONE}, \{c \bmod k\} \rangle$ in an analysis based on mod-k residues. Individual instructions are analyzed as follows:

`load` r`,` *addr* : Our analysis currently doesn't keep track of the contents of memory locations, except for read-only sections of the text and data segments.[3] Thus, if *addr* corresponds to a read-only memory location with

[2] Strictly speaking, the analysis should map each register at each program point to a set of address descriptors. For pragmatic reasons—see Section 3.2.2 for details—we use a widening operation [7] to ensure that at each program point, each register is mapped to a singleton set of address descriptors. For simplicity, we do not distinguish between such a set and the single address descriptor it contains.

[3] Our implementation uses the contents of these read-only sections to obtain global addresses: these include global variables as well as addresses of jump tables and functions called indirectly through function pointers.

contents *val*, then the address descriptor for r is $\langle \text{NONE}, \{\mathit{val} \bmod k\}\rangle$. Otherwise, we can say nothing about the contents of r after the load instruction, so the resulting address descriptor is $\langle I, \{0\}\rangle$.

`store r, addr` : Since a store operation does not affect the contents of any register, this instruction does not have any effect on any address descriptors.

`add` src_a, src_b, *dest* : Let the address descriptors for src_a and src_b immediately before instruction I be $A_a = \langle I_a, X_a\rangle$ and $A_b = \langle I_b, X_b\rangle$ respectively. There are two possibilities:

– If $A_a \not\simeq \perp$, $A_b \not\simeq \perp$, and $I_a = \text{NONE}$ (the situation where $I_b = \text{NONE}$ is symmetric), let $A' = \langle I_b, X'\rangle$, where $X' = \{(x_a + x_b) \bmod k \mid x_a \in X_a, x_b \in X_b\}$. The address descriptor for *dest* is $\langle I, \{0\}\rangle$ if $A' \simeq \perp$, and is A' otherwise.

– Otherwise, we can't say anything about the result of this operation, so the address descriptor for *dest* after I is taken to be $\langle I, \{0\}\rangle$.

The correctness of the first case follows straightforwardly from the rules for mod-k arithmetic [16]; the second case is obviously safe, but merits some discussion: if $A_a \simeq \perp$, $A_b \simeq \perp$, or $I_a \neq I_b$, it's easy to see that we can't say anything about the result of the operation; if $I_a = I_b = I_0$ for some I_0, it's tempting to think that the resulting address descriptor could be given as $\langle I_0, X'\rangle$, where $X' = \{(x_a + x_b) \bmod k \mid x_a \in X_a, x_b \in X_b\}$, but this is not the case. The reason is that in this case, the sets of values denoted by A_a and A_b are:

$$conc_P(\langle I_0, X_a\rangle) = \{w_a + ik + x_a \mid w_a \in val_P(I_0), x_a \in X_a, i \in \mathbf{Z}^+\}; \text{ and}$$
$$conc_P(\langle I_0, X_b\rangle) = \{w_b + ik + x_b \mid w_b \in val_P(I_0), x_b \in X_b, i \in \mathbf{Z}^+\}$$

so the address descriptor A' for *dest* after this instruction should be such that

$$\begin{aligned} conc_P(A') &= \{w_a + w_b + i_1 k + x_a + i_2 k + x_b \mid w_a, w_b \in val_P(I_0), x_a \in X_a, x_b \in X_b, i_1, i_2 \in \mathbf{Z}^+\} \\ &= \{w_a + w_b + ik + x_a + x_b \mid w_a, w_b \in val_P(I_0), x_a \in X_a, x_b \in X_b, i \in \mathbf{Z}^+\}. \end{aligned}$$

However, $\langle I_0, X'\rangle$ clearly does not give this, because it does not account for the fact that the $w \in val_P(I_0)$ component is also added into the result of the `add` instruction:

$$conc_P(\langle I_0, X'\rangle) = \{w + ik + x_a + x_b \mid w \in val_P(I_0), x_a \in X_a, x_b \in X_b, i \in \mathbf{Z}^+\}.$$

`mult` src_a, src_b, *dest* : Let the address descriptors for src_a and src_b immediately before instruction I be $A_a = \langle I_a, X_a\rangle$ and $A_b = \langle I_b, X_b\rangle$ respectively. There are three possibilities:

– If $A_a \not\simeq \perp$, $A_b \not\simeq \perp$, and both I_a and I_b are NONE, let $X_c = \{(x_a \times x_b) \bmod k \mid x_a \in X_a, x_b \in X_b\}$, and $A' = \langle \text{NONE}, X_c\rangle$. The address descriptor for *dest* is $\langle I, \{0\}\rangle$ if $A' \simeq \perp$, and is A' otherwise.

– Otherwise, if $A_a \not\simeq \perp$, $A_b \not\simeq \perp$, and $I_a = \text{NONE}$ (the case where $I_b = \text{NONE}$ is symmetric), let $X_c = \{(x_a \times x_b) \bmod k \mid x_a \in X_a, x_b \in \mathbf{Z}_k\}$, and $A' = \langle \text{NONE}, X_c\rangle$. The address descriptor for *dest* is $\langle I, \{0\}\rangle$ if $A' \simeq \perp$, and is A' otherwise.

– Otherwise, we can't say much about the result of the multiplication, so the address descriptor for *dest* after instruction I is $\langle I, \{0\}\rangle$.

Again, the correctness of the first case follows easily from the rules for mod-k arithmetic; the second case can be thought of as "widening" A_b to $\langle \text{NONE}, \mathbf{Z}_k\rangle$, which is obviously safe, and then applying the first case; the reasoning for the third case is analogous to that for the `add` instruction above.

In typical RISC code, the most commonly encountered address expression by far involves a fixed displacement off a base register, which corresponds to the `add` instruction discussed above. As such it is especially important that this case be handled efficiently. It turns out that given an address descriptor $\langle I, X\rangle$ for reg_a, with X represented as a bit vector, the bit vector X' in the descriptor $\langle I, X' \rangle$ for reg_c can be obtained simply by "rotating up" the bit-vector for X by c bits, and this is easy to implement efficiently. As an example, suppose that $X = \{1, 5, 6\}$ in a mod-8 residue analysis, and $c = 3$, then $X' = \{4, 8, 9\}$ mod $8 = \{4, 0, 1\}$. If we represent these sets as bit vectors with the smallest element on the right, then X = `0110 0010`; rotating up (i.e., to the left) by 3 bits gives us the vector `0001 0011`, which is precisely the bit vector for X'.

3.2.2 Propagating Address Descriptors

Conceptually, if we consider all possible execution paths through a program, each register at each program point will correspond to a set of values; abstracting from this, one would expect an analysis to map each register to a set of address descriptors at each program point. Given the handling of individual instructions as described in the previous section, the analysis is now a conceptually straightforward forward dataflow analysis where we compute the meet-over-all-paths solution,[4] with union as the meet operator [1].

It turns out that if each register, at each program point, is mapped to a set of address descriptors, the memory requirements for the analysis can become excessive for large programs. This is due partly because fully linked executables tend to be considerably larger than source language modules, and partly because reasoning about address arithmetic is usually less precise than, say, reasoning about aliasing at the source level. As a pragmatic measure, therefore, a widening operation [7] is used to ensure that at each program point, each register is mapped to a singleton set of address descriptors—or, equivalently, a single address descriptor. As mentioned in Section 3.1, the set of address descriptors forms a lattice with respect to the precision ordering $\trianglelefteq$. The widening operation $\triangledown$ is defined to be simply the meet operation with respect to $\trianglelefteq$. In effect, what this does is that if a program point B has two predecessors B_0 and B_1, such that the address descriptors for a register r at B_0 and B_1 are $A_0 = \langle I_0, X_0 \rangle$ and $A_1 = \langle I_1, X_1 \rangle$ respectively, where neither A_0 nor A_1 are $\top$, and $I_0 \neq I_1$, then the address descriptor for r at B is $A_0 \triangledown A_1 = \bot$.

This results in a reasonably memory-efficient analysis: for each basic block we need two address descriptors per register, one for the IN set, at the entry to the block, and one for the OUT set, at the exit. Thus, for a given choice of k, the analysis requires $2RN(k + w)$ bits of memory for a program with N basic blocks on a machine with R registers, where w is the number of bits per machine word.[5]

3.3 Reasoning about Alias Relationships

Given two address descriptors $A_1 \equiv \langle I_1, M_1 \rangle$ and $A_2 \equiv \langle I_2, M_2 \rangle$ at two points in a program, under what conditions can we conclude that they definitely do not refer to the same address? If $I_1 \neq I_2$ we cannot say much about any relationship that may hold between A_1 and A_2, and so have to assume that they may refer to the same location. However, it is not sufficient to require that $I_1 = I_2$ and $M_1 \cap M_2 = \emptyset$, since the value computed by a particular instruction may be different when that instruction is executed at different times. The following proposition gives a simple sufficient condition for determining that two address expressions denote disjoint sets of addresses:

Proposition 3.1 *Address descriptors $A_1 \equiv \langle I, M_1 \rangle$ at program point p_1 and $A_2 \equiv \langle I, M_2 \rangle$ at program point p_2 denote disjoint sets of addresses if (i) I dominates both p_1 and p_2; (ii) either p_1 dominates p_2, or p_2 dominates p_1; and (iii) $M_1 \cap M_2 = \emptyset$.*

Proof Conditions (i) and (ii) ensure that both the program points p_1 and p_2 see the same value computed by instruction I. Condition (iii) then ensures that relative to this value, the set of addresses referred to at p_1 is disjoint from that referred to at p_2. □

[4] Since our current implementation is not context-sensitive in its treatment of inter-procedural information flow, a meet-over-all-paths solution suffices; a context-sensitive treatment would have required a meet-over-all-valid-paths solution.

[5] This can be reduced to $RN(k + w)$ bits, as in our implementation, by storing only OUT sets, since the IN set of a block can be computed fairly easily from the OUT sets of its predecessors.

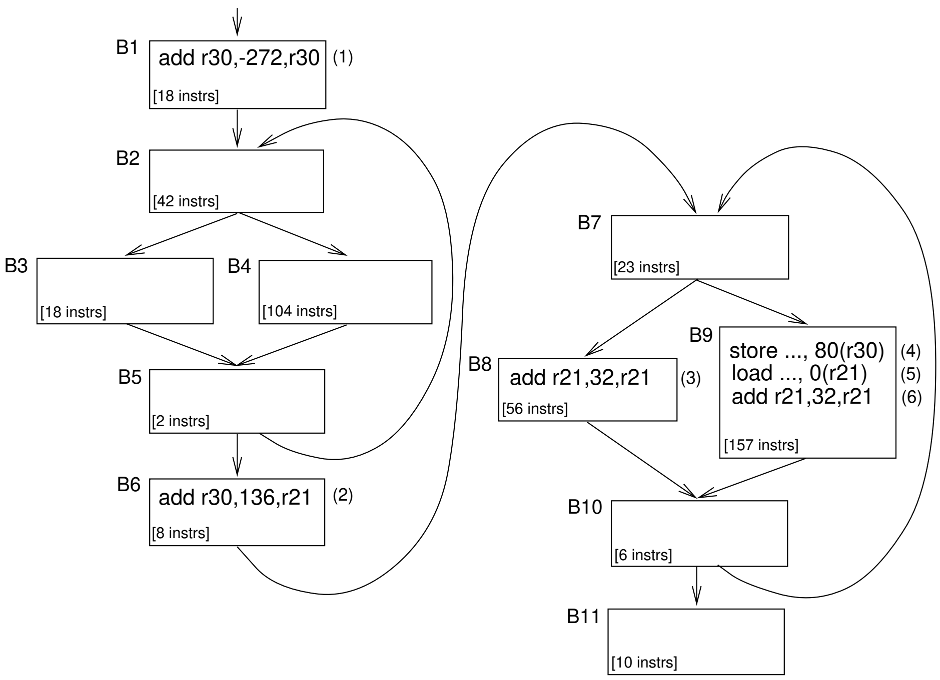

Figure 2: Flowgraph for Example 3.2 [Program: `ijpeg`; function: `jpeg_idct_ifast()`]

Example 3.2 As an example of the application of this analysis to a real program, Figure 2 shows the flow graph of the function `jpeg_idct_ifast()`, from the SPEC benchmark program `ijpeg`, which implements a fast integer inverse discrete cosine transform. To reduce clutter, only a few relevant instructions are shown explicitly: the number in brackets at the lower left hand corner of each basic block indicates the total number of instructions in that basic block. Register `r30` is the stack pointer, while `r21` is used to walk through a local array of structures with a stride of 32 bytes.

Using the current implementation of our analysis, which uses mod-64 residues, the address descriptor for register `r21` immediately after instruction (2) in block B6 is computed as $\langle (1), \{8\} \rangle$, where (1) is the instruction in block B1 that defines the value of `r30`. Each iteration of the loop B7-B8-B9-B10 increments `r21` by 32, so the address descriptor for `r21` on entry to block B9 is $\langle (1), \{8, 40\} \rangle$; however, register `r30` is not changed in the loop, so its address descriptor in B9 is $\langle (1), \{0\} \rangle$. Since the requirements of Proposition 3.1 are trivially satisfied within block B9, we can conclude from this that the store instruction (4), namely, `store ..., 80(r30)`, refers to a different location than instruction (5), namely, `load ..., 0(r21)`. □

4 Experimental Results

We evaluated our analysis on the SPEC-95 benchmarks as well as some non-SPEC applications: `agrep`, a pattern matching utility; `appbt` and `appsp`, computational fluid dynamics codes originally from NASA; `latex`, a popular document formatting tool; and `nucleic2`, a numerical benchmark that finds the 3-dimensional structure of a nucleic acid molecule. The input programs were compiled with the DEC C compiler V5.2-023 invoked as `cc -O4 -Wl,-r -Wl,-d -Wl,-z -non_shared` (for the C programs), and the DEC Fortran compiler version 3.8 invoked as `f77 -O4 -Wl,-r -Wl,-d -Wl,-z -non_shared` (for the Fortran programs), resulting in statically linked executables. The timings were obtained on a DEC Alpha workstation, with a 300 MHz Alpha 21164 processor with 512 Mbytes of main memory, running Digital Unix 4.0. Table 1 shows the precision of the analysis, while Table 2 shows its the time and space requirements. The numbers presented correspond to mod-k residues with $k = 64$ (this choice was de-

PROGRAM	TOTAL	ONE		FEW		TOTAL KNOWN		UNKNOWN	
applu	38973	11083	[28.44%]	5075	[13.02%]	16158	[41.46%]	22814	[58.54%]
apsi	46641	12344	[26.47%]	4930	[10.57%]	17274	[37.04%]	29366	[62.96%]
compress	6375	2070	[32.47%]	235	[3.69%]	2305	[36.16%]	4070	[63.84%]
fpppp	39777	12431	[31.25%]	3726	[9.37%]	16157	[40.62%]	23619	[59.38%]
gcc	137389	44021	[32.04%]	6698	[4.88%]	50719	[36.92%]	86669	[63.08%]
go	31596	7472	[23.65%]	5310	[16.81%]	12782	[40.45%]	18814	[59.55%]
hydro2d	37855	9668	[25.54%]	4711	[12.45%]	14379	[37.98%]	23475	[62.01%]
ijpeg	22179	8473	[38.20%]	1685	[7.60%]	10158	[45.80%]	12021	[54.20%]
li	12466	3919	[31.44%]	307	[2.46%]	4226	[33.90%]	8240	[66.10%]
m88ksim	17516	5271	[30.09%]	651	[3.72%]	5922	[33.81%]	11594	[66.19%]
mgrid	35696	9150	[25.63%]	3840	[10.76%]	12990	[36.39%]	22705	[63.61%]
perl	41039	14777	[36.01%]	1054	[2.57%]	15831	[38.57%]	25208	[61.42%]
su2cor	38052	10434	[27.42%]	4515	[11.87%]	14949	[39.29%]	23103	[60.71%]
swim	34187	9454	[27.65%]	4035	[11.80%]	13489	[39.46%]	20698	[60.54%]
tomcatv	33829	9356	[27.66%]	3905	[11.54%]	13261	[39.20%]	20568	[60.80%]
turb3d	37930	9857	[25.99%]	4187	[11.04%]	14044	[37.03%]	23885	[62.97%]
vortex	59021	19310	[32.72%]	1295	[2.19%]	20605	[34.91%]	38413	[65.08%]
wave5	44047	12113	[27.50%]	7553	[17.15%]	19666	[44.65%]	24381	[55.35%]

(a) SPEC-95 benchmarks

PROGRAM	TOTAL	ONE		FEW		TOTAL KNOWN		UNKNOWN	
agrep	11104	3581	[32.25%]	865	[7.79%]	4446	[40.04%]	6652	[59.91%]
appbt	14582	5353	[36.71%]	3280	[22.49%]	8633	[59.20%]	5948	[40.79%]
appsp	10575	3520	[33.29%]	1886	[17.84%]	5406	[51.12%]	5169	[48.88%]
latex	28765	8673	[30.15%]	2008	[6.98%]	10681	[37.13%]	18083	[62.87%]
nucleic2	25196	14738	[58.49%]	307	[1.22%]	15045	[59.71%]	10151	[40.29%]

(b) Non-SPEC applications

Key: TOTAL : Total no. of load/store instructions [static counts]

ONE : No. of load/store instructions whose mod-k residue set has cardinality 1.

FEW : No. of load/store instructions whose mod-k residue set has cardinality n, $1 < n < k$.

TOTAL KNOWN : ONE+FEW.

UNKNOWN : TOTAL − TOTAL KNOWN.

Table 1: Precision of Analysis (load/store instructions)

PROGRAM	BASIC BLOCKS	INSTRUCTIONS	ANALYSIS TIME (sec)	MEMORY USED (Mbytes)
applu	24939	117247	20.28	9.13
apsi	27334	135270	21.55	10.01
compress	4425	18489	2.93	1.62
fpppp	24778	118183	18.68	9.07
gcc	79037	321986	64.65	28.94
go	15734	74361	12.48	5.76
hydro2d	26048	115957	20.24	9.54
ijpeg	10928	57447	8.96	4.00
li	7856	31572	4.51	2.88
m88ksim	10012	44489	5.48	3.67
mgrid	25025	109260	18.98	9.16
perl	22270	99789	13.86	8.16
su2cor	24827	115547	19.21	9.09
swim	23491	104674	17.66	8.60
tomcatv	23264	103406	17.73	8.52
turb3d	25687	114888	20.51	9.41
vortex	28240	129092	11.26	10.34
wave5	26309	132299	21.50	9.63

(a) SPEC-95 benchmarks

PROGRAM	BASIC BLOCKS	INSTRUCTIONS	ANALYSIS TIME (sec)	MEMORY USED (Mbytes)
agrep	6744	32450	5.65	2.47
appbt	5935	39981	4.96	2.17
appsp	4427	27289	3.48	1.62
latex	14350	66011	8.56	5.26
nucleic2	4090	37078	2.38	1.50

(b) Non-SPEC applications

Table 2: Cost of Analysis

termined in part by the fact that the set of mod-k residues for this choice of k corresponds to a bit vector that fits exactly in one 64-bit machine word), combined with the local analysis described in Section 2.

Precision: Traditionally, the precision of alias analysis algorithms is often presented in terms of the average size of points-to sets or alias sets. In our context, however, there are no points-to or alias sets: a more meaningful measure, perhaps, is the (relative) number of memory references—i.e., load and store instructions—for which the analysis is able to provide information that would not have been available otherwise. This information is presented in Table 1. It can be seen that in the programs tested, the analysis is able to provide information for roughly 35%–60% of the memory reference instructions. Preliminary investigations indicate that much of the loss in precision occurs due to two reasons: first, because we don't keep track of the contents of memory, information about a register will be lost if it is saved to memory and subsequently restored; and second, the widening operation described in Section 3.2.2 causes information to be lost if a register can have different defining instructions at different predecessors of a join point in the control flow graph.

Cost: Table 2 gives the time and space costs of our analysis. Columns 2 and 3 give the size of each benchmark, measured, respectively, in the total number of basic blocks and instructions in the program, measured after the elimination

Program	Total Loads ($\times 10^6$) (tot)	Deletable ($\times 10^6$) (del)	del/tot (%)
appbt	210.75	11.41	5.4
appsp	108.32	2.29	2.1
fpppp	41828.25	17111.94	40.9
m88ksim	15209.48	197.12	1.3
nucleic2	94.63	4.74	5.0
su2cor	7405.70	212.51	2.9
vortex	22989.38	531.60	2.3
wave5	7728.41	446.05	5.8

Table 3: Utility of Analysis: Deletion of unnecessary `load` instructions

of dead and unreachable code. Column 4 then gives the total analysis time in seconds, while column 5 gives the total memory requirements of the analysis in Mbytes. The analysis times range from about 2 seconds to 20 seconds, with the `gcc` program an outlier with a total analysis time of a little over a minute. These numbers are somewhat higher than we would like, but the reason for this is that every instruction within a basic block is examined whenever that basic block is processed. As Figure 3 indicates, the time taken to analyze a program in practice varies essentially linearly as the number of instructions in the program. The memory requirement of the analysis typically varies from about 1.5 Mbytes to 10 Mbytes, with `gcc` having a high requirement of about 29 Mbytes. Because of the widening operation described in Section 3.2.2, the memory requirements of the analysis are linear in the number of basic blocks in the input program: we feel that this is essential if the analysis is to be usable for large programs.

Utility: The only optimization for which we have had the time to evaluate the utility of our alias analysis at this time involves the elimination of unnecessary `load` instructions. Preliminary results are shown in Table 3, which gives dynamic counts of the number of load instructions that can be removed. Since, at optimization level `-O4`, global register allocation had already been carried out by the compiler, we were pleasantly surprised that our analysis could still detect a significant number of `load` instructions that could potentially be eliminated. Our system currently removes many of these `loads`, and we are working on other optimizations that will free up additional registers that can then be used for this purpose. We also plan to incorporate the results of alias analysis into our instruction scheduler as well as a number other optimizations, and expect to have more extensive experimental results for the utility of this information shortly.

5 Related Work

While a number of systems have been described for link-time code optimization [5, 14, 15, 26, 29, 30, 32], to the best of our knowledge none of these carry out any alias analysis on the executable files they process.

There is an extensive body of work on pointer alias analysis of various kinds (see, for example, [2, 3, 4, 6, 8, 10, 11, 12, 13, 17, 18, 20, 21, 22, 23, 25, 27, 28, 31, 33, 34]). The work most closely related to ours is that of Wilson and Lam [34], who describe a low-level pointer alias analysis for C programs. Their work attempts to deal with “nasty” features of real programs and can handle simple pointer increments and decrements, but is unable to cope with the more complex address arithmetic common in executable code (see Example 3.2). Also, it restricts itself to C language features, and so cannot handle arithmetic arising from idiosyncracies of other languages, e.g., manipulation of pointers with “tag bits,” that may be encountered in executable code. Their algorithm is context-sensitive at the inter-procedural level, however, while our current implementation is context-insensitive (conceptually, it would not be too difficult to obtain a context-sensitive version of our algorithm, but we have not had time to implement this yet). The remaining analyses cited are all high level analyses that typically disregard type casts, pointer arithmetic, out-of-bounds array accesses, etc. As argued earlier, such analyses are of limited utility at the machine code level.

Also related is the work on dependence analysis in the scientific computing literature (see, for example, [35, 36]).

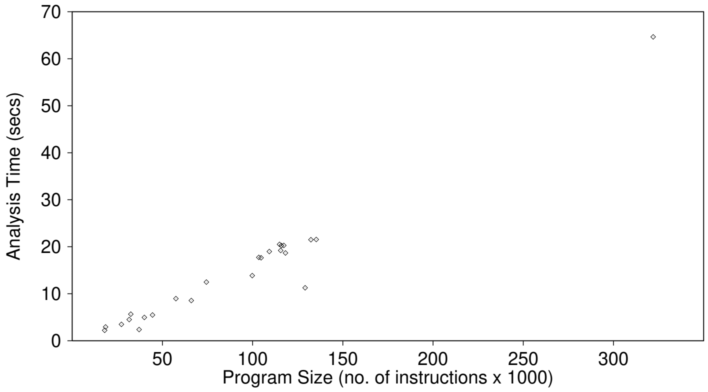

Figure 3: Variation of analysis time with input size

While the goals of this work are conceptually similar to ours—namely, disambiguating array references whose indices can involve arithmetic expressions—the algorithms used for dependence analysis are very different from that described here. Since dependence analysis is typically formulated as a source level intra-procedural analysis, the analysis problems tend to be relatively small in size. Because of this, dependence analyses are able to use relatively more sophisticated, but also more expensive, algorithms than ours. We do not know of any attempts to apply such algorithms for whole-program analysis, and it is not obvious to us that the algorithms involved would scale up to problems of this size.

6 Conclusions

Recent years have seen increasing interest in reasoning about and manipulating executable files. Such manipulations can benefit greatly from information about aliasing. Unfortunately, there is a fundamental mismatch between the features present in executable programs and the features handled by existing pointer alias analyses: such analyses are typically formulated in terms of source-level constructs, and do not handle features such as pointer arithmetic and out-of-bound array references, whereas these are precisely the features encountered in executable programs. This paper describes a simple algorithm that can handle these features, and which can be used for alias analysis of executable programs. In order to be practical, the algorithm is careful to keep its memory requirements low, sacrificing precision where necessary to achieve this goal. Experimental results indicate that it is nevertheless able to provide nontrivial information about roughly 35%–60% of the memory references across a variety of benchmark programs.

References

[1] A. V. Aho, R. Sethi and J. D. Ullman, *Compilers – Principles, Techniques and Tools*, Addison-Wesley, 1986.

[2] M. Burke, P. Carini, J. D. Choi, and M. Hind, "Flow-insensitive interprocedural alias analysis in the presence of pointers", in *Languages and Compilers for Parallel Computing: Proceedings of the 7th International Workshop*, eds. K. Pingali, U. Bannerjee, D. Gelernter, A. Nicolau and D. Padua, Aug. 1994. Springer-Verlag LNCS vol. 892, pp. 234–250.

[3] D. R. Chase, M. Wegman, and F. K. Zadeck, "Analysis of Pointers and Structures", *Proc. SIGPLAN '90 Conference on Programming Language Design and Implementation*, June 1990, pp. 296–310.

[4] J.-D. Choi, M. Burke, and P. Carini, "Efficient Flow-Sensitive Interprocedural Computation of Pointer-Induced Aliases and Side Effects", *Proc. 20th ACM Symposium on Principles of Programming Languages*, Jan. 1993, pp. 232–245.

[5] R. Cohn, D. Goodwin, P. G. Lowney, and N. Rubin, "Spike: An Optimizer for Alpha/NT Executables", *Proc. USENIX Windows NT Workshop*, Aug. 1997.

[6] K. D. Cooper and K. Kennedy, "Fast Interprocedural Alias Analysis", *Proc. 16th ACM Symposium on Principles of Programming Languages*, Jan. 1989, pp. 49–59.

[7] P. Cousot and R. Cousot, "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Apporoximation of Fixpoints", *Proc. Fourth ACM Symposium on Principles of Programming Languages*, 1977, pp. 238-252.

[8] D. Coutant, "Retargetable High-Level Alias Analysis", *Proc. 13th ACM Symposium on Principles of Programming Languages*, Jan. 1986, pp. 110–118.

[9] K. De Bosschere and S. K. Debray, "`alto`: A Link-Time Optimizer for the DEC Alpha", Technical Report 96-15, Dept. of Computer Science, The University of Arizona, June 1996.

[10] A. Deutsch, "On determining lifetime and aliasing of dynamically allocated data in higher-order functional specifications", *Proc. 17th ACm Symposium on Principles of Programming Languages*, Jan. 1990, pp. 157–168.

[11] A. Deutsch, "Interprocedural May-Alias Analysis for Pointers: Beyond k-limiting", *Proc. SIGPLAN '94 Conference on Programming Language Design and Implementation*, June 1994, pp. 230–241.

[12] A. Diwan, K. S. McKinley and J. E. B. Moss, "Type-Based Alias Analysis", Manuscript, Dept. of Computer Science, University of Massachusetts, Amherst, 1996.

[13] M. Emami, R. Ghiya and L. J. Hendren, "Context-Sensitive Interprocedural Points-to Analysis in the Presence of Function Pointers", *Proc. SIGPLAN '94 Conference on Programming Language Design and Implementation*, June 1994, pp. 242–256.

[14] M. F. Fernández, "Simple and Effective Link-Time Optimization of Module-3 Programs", *Proc. SIGPLAN '95 Conference on Programming Language Design and Implementation*, June 1995, pp. 103–115.

[15] D. W. Goodwin, "Interprocedural Dataflow Analysis in an Executable Optimizer", *Proc. SIGPLAN '97 Conference on Programming Language Design and Implementation*, June 1997, pp. 122–133.

[16] R. L. Graham, D. E. Knuth, and O. Patashnik, *Concrete Mathematics*, Addison-Wesley, 1989.

[17] S. Horwitz, P. Pfeiffer, and T. Reps, "Dependence Analysis for Pointer Variables", *Proc. SIGPLAN '89 Conference on Programming Language Design and Implementation*, June 1989, pp. 28–40.

[18] J. Hummel, L. J. Hendren, and A. Nicolau, "A General Data Dependence Test for Dynamic, Pointer-Based Data Structures", *Proc. SIGPLAN '94 Conference on Programming Language Design and Implementation*, June 1994, pp. 218–229.

[19] M. S. Johnson and T. C. Miller, "Effectiveness of a Machine-Level Global Optimizer", *Proc. SIGPLAN '86 Symposium on Compiler Construction*, June 1986, pp. 99–108.

[20] N. D. Jones and S. S. Muchnick, "Flow analysis and optimization of LISP-like structures", in *Program Flow Analysis*, eds. S. S. Muchnick and N. D. Jones, Prentice Hall, 1981, pp. 102–131.

[21] N. D. Jones and S. S. Muchnick, "A flexible approach to interprocedural data flow analysis and programs with recursive data structures", *Proc. 9th ACM Symposium on Principles of Programming Languages*, Jan. 1982, pp. 66–74

[22] W. Landi and B. G. Ryder, "Pointer-induced Aliasing: A Problem Classification", *Proc. 18th ACM Symposium on Principles of Programming Languages*, Jan. 1991, pp. 93–103.

[23] W. Landi and B. G. Ryder, "A Safe Approximate Algorithm for Interprocedural Pointer Aliasing", *Proc. SIGPLAN '92 Conference on Programming Language Design and Implementation*, June 1992, pp. 235–248.

[24] J. R. Larus and E. Schnarr, "EEL: Machine-independent Executable Editing", *Proc. SIGPLAN '95 Conference on Programming Language Design and Implementation*, June 1995, pp. 291–300.

[25] J. R. Larus and P. N. Hilfinger, "Detecting Conflicts Between Structure Accesses", *Proc. SIGPLAN '88 Conference on Programming Language Design and Implementation*, June 1988, pp. 21–34.

[26] T. Romer, G. Voelker, D. Lee, A. Wolman, W. Wong, H. Levy, B. N. Bershad, and J. B. Chen, "Instrumentation and Optimization of Win32/Intel Executables", 1997 USENIX Windows NT Workshop (to appear).

[27] E. Ruf, "Context-Insensitive Alias Analysis Reconsidered", *Proc. SIGPLAN '95 Conference on Programming Language Design and Implementation*, June 1995, pp. 13–22.

[28] M. Shapiro and S. Horwitz, "Fast and Accurate Flow-Insensitive Points-To Analysis", *Proc. 24th. ACM Symposium on Principles of Programming Languages*, Jan. 1997, pp. 1–14.

[29] A. Srivastava and D. W. Wall, "A Practical System for Intermodule Code Optimization at Link-Time", *Journal of Programming Languages*, pp. 1–18, March 1993.

[30] A. Srivastava and D. W. Wall, "Link-time Optimization of Address Calculation on a 64-bit Architecture", *Proc. SIGPLAN '94 Conference Programming Language Design and Implementation*, June 1994, pp. 49–60.

[31] B. Steensgaard, "Points-to Analysis in Almost Linear Time", *Proc. 23th. ACM Symposium on Principles of Programming Languages*, Jan. 1996, pp. 32–41

[32] D. W. Wall, "Global Register Allocation at Link Time", *Proc. SIGPLAN '86 Symposium on Compiler Construction*, July 1986, pp. 264–275.

[33] W. E. Weihl, "Interprocedural data flow analysis in the presence of pointers, procedure variables, and label variables", *Proc. ACM Symposium on Principles of Programming Languages*, Jan. 1980, pp. 83–94.

[34] R. P. Wilson and M. S. Lam, "Efficient Context-Sensitive Pointer Analysis for C Programs", *Proc. SIGPLAN '95 Conference on Programming Language Design and Implementation*, June 1995, pp. 1–12.

[35] M. Wolfe, *Optimizing Supercompilers for Supercomputers*, MIT Press, Cambridge, Mass., 1989.

[36] H. Zima and B. Chapman, *Supercompilers for Parallel and Vector Computers*, ACM Press, New York, 1991.