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. 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 $r_{16}$ and $r_{17}$, 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 $\text{load } \text{reg}_a, k(\text{reg}_b)$ and $\text{store } \text{reg}_a, k(\text{reg}_b)$, where $k$ is a constant, and have the effect of reading from, or writing to, the location whose address is $k + \text{contents of(}\text{reg}_b\text{)}$. To model arithmetic we assume the instructions $\text{add } \text{src}_1, \text{src}_2, \text{dest}$ and $\text{mul } \text{src}_1$.

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**

```c
int g(int *x, int *y)
{
    *x = 1;
    *y = 0;
    ... 
}
```

**Executable Code**

```assembly
# arg1 in r16, arg2 in r17
add r30, -32, r30     # allocate stack frame
store r26, 0(r30)      # save return address
store 1, 0(r16)        # save return address
store 0, 0(r17)
... 
```

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

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): ..

\[
\begin{align*}
    l_1 &: \text{add } r_1, c_1, r_2; \\
    l_2 &: \text{add } r_2, c_2, r_3; \ldots; \\
    l_k &: \text{add } r_k, c_k, r \\
    l'_1 &: \text{add } r'_1, c'_1, r'_2; \\
    l'_2 &: \text{add } r'_2, c'_2, r'_3; \ldots; \\
    l'_m &: \text{add } r'_m, c'_m, r'
\end{align*}
\]

where $k, m \geq 0$, such that (i) $l_j$ uses the definition of $r_j$ in $l_{j-1}$, and $l'_j$ uses the definition of $r'_j$ in $l'_{j-1}$; (ii) either both $l_1$ and $l'_1$ use the same definition of $r_1$ in the block $B$, or neither use any definition of $r_1$ in $B$; and (iii) $\sum_{i=0}^{k} c_i \neq \sum_{i=0}^{m} c'_i$. Then, the value of register $r$ immediately after instruction $l_k$ is different from that of register $r'$ immediately
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 $\mathbb{Z}_k = \{0, \ldots, k-1\}$. 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 $r(x)$, 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 {NONE, 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:

$$\text{conc}_P(\langle I, X \rangle) = \{ w + ik + x \mid w \in val_P(I), x \in X, i \in \mathbb{Z}^+ \}.$$  

The relative precision of different address descriptors can be characterized via the binary relation $\preceq$:
**Definition 3.2** An address descriptor \( \langle l_2, x_2 \rangle \) is more precise than a descriptor \( \langle l_1, x_1 \rangle \), written \( \langle l_1, x_1 \rangle \leq \langle l_2, x_2 \rangle \), if and only if (i) \( l_1 = \text{ANY} \) or \( x_1 = \mathbb{Z}_k \); or (ii) \( x_2 = \emptyset \); or (iii) \( l_1 = l_2 \) and \( x_2 \subseteq x_1 \).

It is straightforward to show that \( \leq \) 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 \( A_1 \leq A_2 \) and \( A_2 \leq A_1 \)—it is easy to show that this is an equivalence relation—and consider the quotient of \( \leq \) 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 \( \leq \) to refer to the resulting partial order. In particular, the equivalence class containing \( \langle l, \mathbb{Z}_k \rangle \) for all \( l \), as well as \( \langle \text{ANY}, M \rangle \) for all \( M \), denotes a total lack of information, and is written as \( \bot \); the equivalence class containing \( \langle l, \emptyset \rangle \) for all \( l \), 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 \( \langle l, M \rangle \) at a program point, we will sometimes abuse terminology and refer to instruction \( l \) 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:

\[
\begin{align*}
\text{store 1, 0(r16)} & \quad (3) \\
\text{store 0, 0(r17)} & \quad (4)
\end{align*}
\]

Assuming that the only call site for \( f() \) is that in \( f() \), we can use instruction (6) as a defining instruction for \( r_{30} \), and thence for \( r_{16} \) and \( r_{17} \)—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:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Address Expression</th>
<th>Address Descriptor</th>
</tr>
</thead>
<tbody>
<tr>
<td>(3)</td>
<td>0(r16)</td>
<td>( \langle {6}, {16} \rangle ) (from instruction (10))</td>
</tr>
<tr>
<td>(4)</td>
<td>0(r17)</td>
<td>( \langle {6}, {20} \rangle ) (from instruction (11))</td>
</tr>
</tbody>
</table>

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. \( \square \)

### 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 \( l \) with destination register \( r \), if we can’t say anything about the value of \( r \) after instruction \( l \), then instead of setting the address descriptor for \( r \) to \( \bot \), we use \( l \) as the defining instruction for \( r \) and associate the address descriptor \( \langle l, \emptyset \rangle \) with \( r \) at the point immediately after \( l \). To simplify the discussion, we assume that an immediate operand \( e \) yields an address descriptor \( \langle \text{NONE}, \{e \mod k\} \rangle \) in an analysis based on mod-\( k \) residues. Individual instructions are analyzed as follows:

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

\( ^2 \)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.
In typical RISC code, the most commonly encountered address expression by far involves a fixed displacement off a src mult store k.
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 $A_a \not\simeq \bot$, $A_b \not\simeq \bot$, or $l_a \neq l_b$, it’s easy to see that we can’t say anything about the result of the operation; if $l_a = l_b = l_0$ for some $l_0$, it’s tempting to think that the resulting address descriptor could be given as $\langle l_0, X' \rangle$, where $X' = \{(x_a + x_b) \mod k \mid x_a \in X_a, x_b \in X_b \}$, 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:

$$
\text{conc}_P(\langle l_0, X_0 \rangle) = \{w_a + i + x_a \mid w_a \in \text{val}_P(l_0), x_a \in X_a, i \in \mathbb{Z}^+\};$
$$
$$
\text{conc}_P(\langle l_0, X_0 \rangle) = \{w_b + i + x_b \mid w_b \in \text{val}_P(l_0), x_b \in X_b, i \in \mathbb{Z}^+\}.$$ 

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

$$
\text{conc}_P(\langle l_0, X' \rangle) = \{w + i + x_a + x_b \mid w \in \text{val}_P(l_0), x_a \in X_a, x_b \in X_b, i \in \mathbb{Z}^+\}.
$$

However, $\langle l_0, X' \rangle$ clearly does not give this, because it does not account for the fact that the $w \in \text{val}_P(l_0)$ component is also added into the result of the add instruction:

$$
\text{conc}_P(\langle l_0, X' \rangle) = \{w + i + x_a + x_b \mid w \in \text{val}_P(l_0), x_a \in X_a, x_b \in X_b, i \in \mathbb{Z}^+\}.
$$

mult src_a, src_b, dest : Let the address descriptors for src_a and src_b immediately before instruction $l$ be $A_a = \langle l_a, X_a \rangle$ and $A_b = \langle l_b, X_b \rangle$ respectively. There are three possibilities:

- If $A_a \not\simeq \bot$, $A_b \not\simeq \bot$, and both $l_a$ and $l_b$ are NONE, let $X_c = \{(x_a \times x_b) \mod k \mid x_a \in X_a, x_b \in X_b \}$, and $A' = \langle \text{NONE}, X_c \rangle$. The address descriptor for dest is $\langle I, \{0\} \rangle$ if $A' \simeq \bot$ and is $A'$ otherwise.

- Otherwise, if $A_a \not\simeq \bot$, $A_b \not\simeq \bot$, and $l_a = \text{NONE}$ (the case where $l_0 = \text{NONE}$ is symmetric), let $X_c = \{(x_a \times x_b) \mod k \mid x_a \in X_a, x_b \in \mathbb{Z}_k \}$, and $A' = \langle \text{NONE}, X_c \rangle$. The address descriptor for dest is $\langle I, \{0\} \rangle$ if $A' \simeq \bot$ and is $A'$ otherwise.

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

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 $\langle \text{NONE}, \mathbb{Z}_k \rangle$, 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 $reg_e$ can be obtained simply by “rotating up” the bit-vector for $X$ by $e$ bits, and this is easy to implement efficiently. As an example, suppose that $X = \{1, 5, 6\}$ in a mod-8 residue analysis, and $e = 3$, 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 $X = 0110 0010; \text{rotating up (i.e., to the left)}$ 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, 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 $\preceq$. The widening operation $\vee$ is defined to be simply the meet operation with respect to $\preceq$. 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 $A_0 \vee A_1 = \bot$.

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.

### 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 $A_1 \equiv \langle I_1, M_1 \rangle$ at program point $p_1$ and $A_2 \equiv \langle I_2, M_2 \rangle$ 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 $p_2$. □

---

4Since 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.

5This 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.
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 jpeg, 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 \((1), \{8\}\), 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 \((1), \{8, 40\}\); however, register r30 is not changed in the loop, so its address descriptor in B9 is \((1), \{0\}\). Since the requirements of Proposition 3.1 are trivially satisfied within block B9, we can conclude from this that the store instruction (4), namely, \textit{store ...}, \textit{80(r30)}, refers to a different location than instruction (5), namely, \textit{load ...}, \textit{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 \texttt{cc -O4 -Wl,-r -Wl,-d -Wl,-z -non_shared} (for the C programs), and the DEC Fortran compiler version 3.8 invoked as \texttt{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-
<table>
<thead>
<tr>
<th>Program</th>
<th>Total</th>
<th>One</th>
<th>Few</th>
<th>Total Known</th>
<th>Unknown</th>
</tr>
</thead>
<tbody>
<tr>
<td>applu</td>
<td>38973</td>
<td>11083 [28.44%]</td>
<td>5075 [13.02%]</td>
<td>16158 [41.46%]</td>
<td>22814 [58.54%]</td>
</tr>
<tr>
<td>apsi</td>
<td>46641</td>
<td>12344 [26.47%]</td>
<td>4930 [10.57%]</td>
<td>17274 [37.04%]</td>
<td>29366 [62.96%]</td>
</tr>
<tr>
<td>compress</td>
<td>6375</td>
<td>2070 [32.47%]</td>
<td>235  [3.69%]</td>
<td>2305 [36.16%]</td>
<td>4070 [63.84%]</td>
</tr>
<tr>
<td>fppp</td>
<td>39777</td>
<td>12431 [31.25%]</td>
<td>3726 [9.37%]</td>
<td>16158 [41.46%]</td>
<td>23619 [59.38%]</td>
</tr>
<tr>
<td>gcc</td>
<td>137389</td>
<td>44021 [32.04%]</td>
<td>6698 [4.88%]</td>
<td>50719 [36.92%]</td>
<td>86669 [63.08%]</td>
</tr>
<tr>
<td>go</td>
<td>31596</td>
<td>7472 [23.65%]</td>
<td>5310 [16.81%]</td>
<td>12782 [40.45%]</td>
<td>18814 [59.55%]</td>
</tr>
<tr>
<td>hydro2d</td>
<td>37855</td>
<td>9668 [25.54%]</td>
<td>4711 [12.45%]</td>
<td>14379 [37.98%]</td>
<td>23475 [62.01%]</td>
</tr>
<tr>
<td>ijpeg</td>
<td>22179</td>
<td>8473 [38.20%]</td>
<td>1685 [7.60%]</td>
<td>10158 [45.80%]</td>
<td>12021 [54.20%]</td>
</tr>
<tr>
<td>li</td>
<td>12466</td>
<td>3919 [31.44%]</td>
<td>307  [2.46%]</td>
<td>4226 [33.90%]</td>
<td>8240 [66.10%]</td>
</tr>
<tr>
<td>m88ksim</td>
<td>17516</td>
<td>5271 [30.09%]</td>
<td>651  [3.72%]</td>
<td>5922 [33.81%]</td>
<td>11594 [66.19%]</td>
</tr>
<tr>
<td>mgrid</td>
<td>35696</td>
<td>9150 [25.63%]</td>
<td>3840 [10.76%]</td>
<td>12990 [36.39%]</td>
<td>22705 [63.61%]</td>
</tr>
<tr>
<td>perl</td>
<td>41039</td>
<td>14777 [36.01%]</td>
<td>1054 [2.57%]</td>
<td>15831 [38.57%]</td>
<td>25208 [61.42%]</td>
</tr>
<tr>
<td>su2cor</td>
<td>38052</td>
<td>10434 [27.42%]</td>
<td>4515 [11.87%]</td>
<td>14949 [39.29%]</td>
<td>23103 [60.71%]</td>
</tr>
<tr>
<td>swim</td>
<td>34187</td>
<td>9454 [27.65%]</td>
<td>4035 [11.80%]</td>
<td>13489 [39.46%]</td>
<td>20698 [60.54%]</td>
</tr>
<tr>
<td>tomcatv</td>
<td>33829</td>
<td>9356 [27.66%]</td>
<td>3905 [11.54%]</td>
<td>13261 [39.20%]</td>
<td>20568 [60.80%]</td>
</tr>
<tr>
<td>turb3d</td>
<td>37930</td>
<td>9857 [25.99%]</td>
<td>4187 [11.04%]</td>
<td>14044 [37.03%]</td>
<td>23885 [62.97%]</td>
</tr>
<tr>
<td>vortex</td>
<td>59021</td>
<td>19310 [32.72%]</td>
<td>1295 [2.19%]</td>
<td>20605 [34.91%]</td>
<td>38413 [65.08%]</td>
</tr>
<tr>
<td>wave5</td>
<td>44047</td>
<td>12113 [27.50%]</td>
<td>7553 [17.15%]</td>
<td>19666 [44.65%]</td>
<td>24381 [55.35%]</td>
</tr>
</tbody>
</table>

(a) SPEC-95 benchmarks

<table>
<thead>
<tr>
<th>Program</th>
<th>Total</th>
<th>One</th>
<th>Few</th>
<th>Total Known</th>
<th>Unknown</th>
</tr>
</thead>
<tbody>
<tr>
<td>agrep</td>
<td>11104</td>
<td>3581 [32.25%]</td>
<td>865  [7.79%]</td>
<td>4446 [40.04%]</td>
<td>6652 [59.91%]</td>
</tr>
<tr>
<td>appbt</td>
<td>14582</td>
<td>5353 [36.71%]</td>
<td>3280 [22.49%]</td>
<td>8633 [59.20%]</td>
<td>5948 [40.79%]</td>
</tr>
<tr>
<td>appsp</td>
<td>10575</td>
<td>3520 [33.29%]</td>
<td>1886 [17.84%]</td>
<td>5406 [51.12%]</td>
<td>5169 [48.88%]</td>
</tr>
<tr>
<td>latex</td>
<td>28765</td>
<td>8673 [30.15%]</td>
<td>2008 [6.98%]</td>
<td>10681 [37.13%]</td>
<td>18083 [62.87%]</td>
</tr>
<tr>
<td>nucleic2</td>
<td>25196</td>
<td>14738 [58.49%]</td>
<td>307  [1.22%]</td>
<td>15045 [59.71%]</td>
<td>10151 [40.29%]</td>
</tr>
</tbody>
</table>

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

(a) SPEC-95 benchmarks

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

(b) Non-SPEC applications

Table 2: Cost of Analysis

terminered 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
d graph.

**Cost:** Table 2 gives the time and space costs of our analysis. Columns 2 and 3 give the size of each benchmark, mea-
sured, respectively, in the total number of basic blocks and instructions in the program, measured after the elimination
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 be potentially 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 of 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 idiosyncrasies 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]).

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

<table>
<thead>
<tr>
<th>Program</th>
<th>Total Loads ($\times 10^4$) (TOT)</th>
<th>Deletable ($\times 10^4$) (DEL)</th>
<th>DEL/TOT (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>appbt</td>
<td>210.75</td>
<td>11.41</td>
<td>5.4</td>
</tr>
<tr>
<td>appsp</td>
<td>108.32</td>
<td>2.29</td>
<td>2.1</td>
</tr>
<tr>
<td>fpppp</td>
<td>41828.25</td>
<td>17111.94</td>
<td>40.9</td>
</tr>
<tr>
<td>m88ksim</td>
<td>15209.48</td>
<td>197.12</td>
<td>1.3</td>
</tr>
<tr>
<td>nucleic2</td>
<td>94.63</td>
<td>4.74</td>
<td>5.0</td>
</tr>
<tr>
<td>su2cor</td>
<td>7405.70</td>
<td>212.51</td>
<td>2.9</td>
</tr>
<tr>
<td>vortex</td>
<td>22989.38</td>
<td>531.60</td>
<td>2.3</td>
</tr>
<tr>
<td>wave5</td>
<td>7728.41</td>
<td>446.05</td>
<td>5.8</td>
</tr>
</tbody>
</table>

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.
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


