Alias Analysis of Executable Code

Saumya Debray, Robert Muth, Matthew Weippert

University of Arizona

Introduction to Alias Analysis

- Alias analysis is a technique used to determine whether a storage location may be accessed in more than one way
- Two pointers are said to be aliased if they point to the same location
- Two memory references are said to have a *may-alias* relation if their aliasing is unknown
- Usually, alias analysis is done on high-level representation in compilers

Alias Analysis Example

```
p.foo = 1;
q.foo = 2;
i = p.foo + 3;
```

In the code snippet, there are three possible alias cases here:

1. The variable p and q cannot alias
2. The variable p and q must alias
3. It cannot be conclusively determined whether p and q alias or not

Alias Analysis Example

```
p.foo = 1;
q.foo = 2;
i = p.foo + 3;
```

If p and q cannot alias, then i = p.foo + 3; can be changed to i = 4. If p and q must alias, then it can be changed to i = 5. In both cases, optimizations can be performed from the alias knowledge.

On the other hand, if it is not known if p and q alias or not, then no optimizations can be performed and the whole code must be executed to get the result

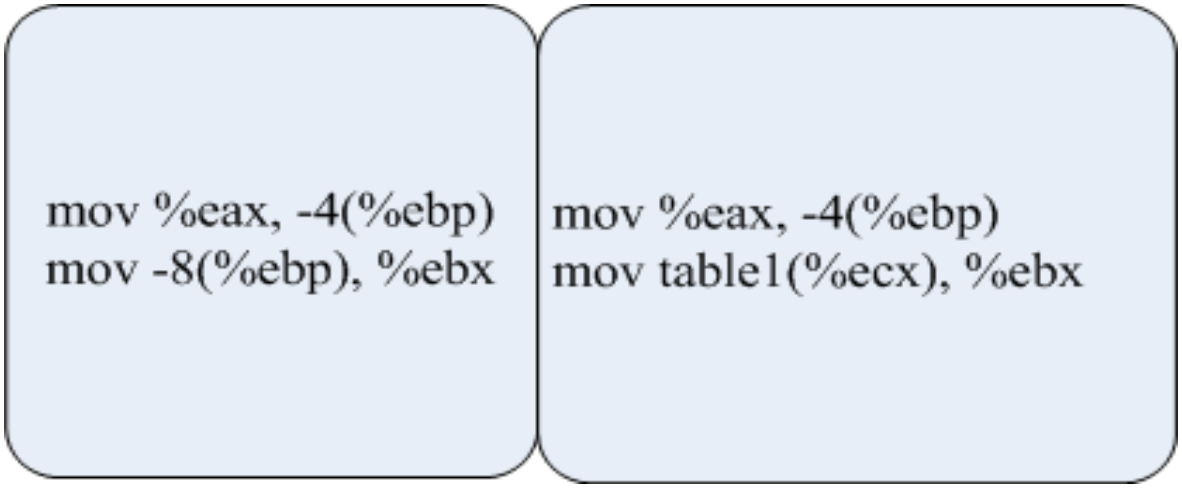
Alias analysis on executable code vs on high-level code

- Disadvantages: no type information, nasty features such as type casts, pointer arithmetic, out-of-bounds array accesses and so on
- Advantages: global information

Local Alias Analysis

- Instruction inspection
  - 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

Instruction Inspection Examples



- In the code snippet on the left, two memory references use different offsets
- In the code snippet on the right, two memory references use different base addresses

Local Alias Analysis

- 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 >= 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) $\sum_{i=1}^{k} c_i \neq \sum_{i=0}^{k} c_i'$
- Then, the value of register r immediately after instruction I_k is different from that of register r' immediately after instruction I_k'.

Global Alias Analysis: Mod-k Residues

- An alias analysis will associate each register with a set of possible addresses at each program point
- Addresses are represented by their mod-k residues, where $k = 2^m$.
- An address descriptor is a pair <I, M>, 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 =<I, M>, the instruction I is said to be the defining instruction of A, while M is called the residue set of A.

Concretization of address descriptor

$$conc_p(\langle I, X \rangle) = \{w + ik + x \mid w \in val_p(I), x \in X, i \in Z^+\}$$

The concretization of A = <I, X>

Relative precision

- The relative precision of different address descriptors can be characterized via the binary relation $\trianglelefteq$
- Definition: An address descriptor $<I_2, X_2>$ is more precise than a descriptor $<I_1, X_1>$, written $<I_1, X_1> \trianglelefteq <I_2, X_2>$, if and on $\trianglelefteq$ f (i) I_1 = ANY or $X_1 = Z_k$; or (ii) $X_2 = \Phi$; or (iii) $I_1 = I_2$ and

$$X_2 \subseteq X_1$$

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

Address Descriptor Example

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))

- The address descriptor for instructions (3) and (4) are <(6), {16}> and <(6), {20}> respectively

The Analysis Algorithm

- The algorithm does not track the contents of memory locations, except for read-only sections of the text and data segments
- Load r, addr. If addr corresponds to a read-only memory location with contents val, then the address descriptor for r is <NONE, {val mod k}>. Otherwise, the resulting address descriptor is <I, {0}>

The Analysis Algorithm

`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 $A_a \not\simeq \bot$, $A_b \not\simeq \bot$, 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 $A' \simeq \bot$, 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 Analysis Algorithm

`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 $A_a \not\simeq \bot$, $A_b \not\simeq \bot$, 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 $A' \simeq \bot$, and is A' otherwise.
- Otherwise, if $A_a \not\simeq \bot$, $A_b \not\simeq \bot$, 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 $A' \simeq \bot$, and is A' otherwise.
- Otherwise, we can't say much about the result of the multiplication, so the address descriptor for *dest* after instruction I is $\langle I, \{0\} \rangle$.

Propagating Address Descriptors

- Due to the complexity of representation of mapping one register to a set of address descriptors at each program point, each register is mapped to a single address descriptor
- 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 T, and $I_0 \neq I_1$, then the address descriptor for r at B is $\perp$

Example

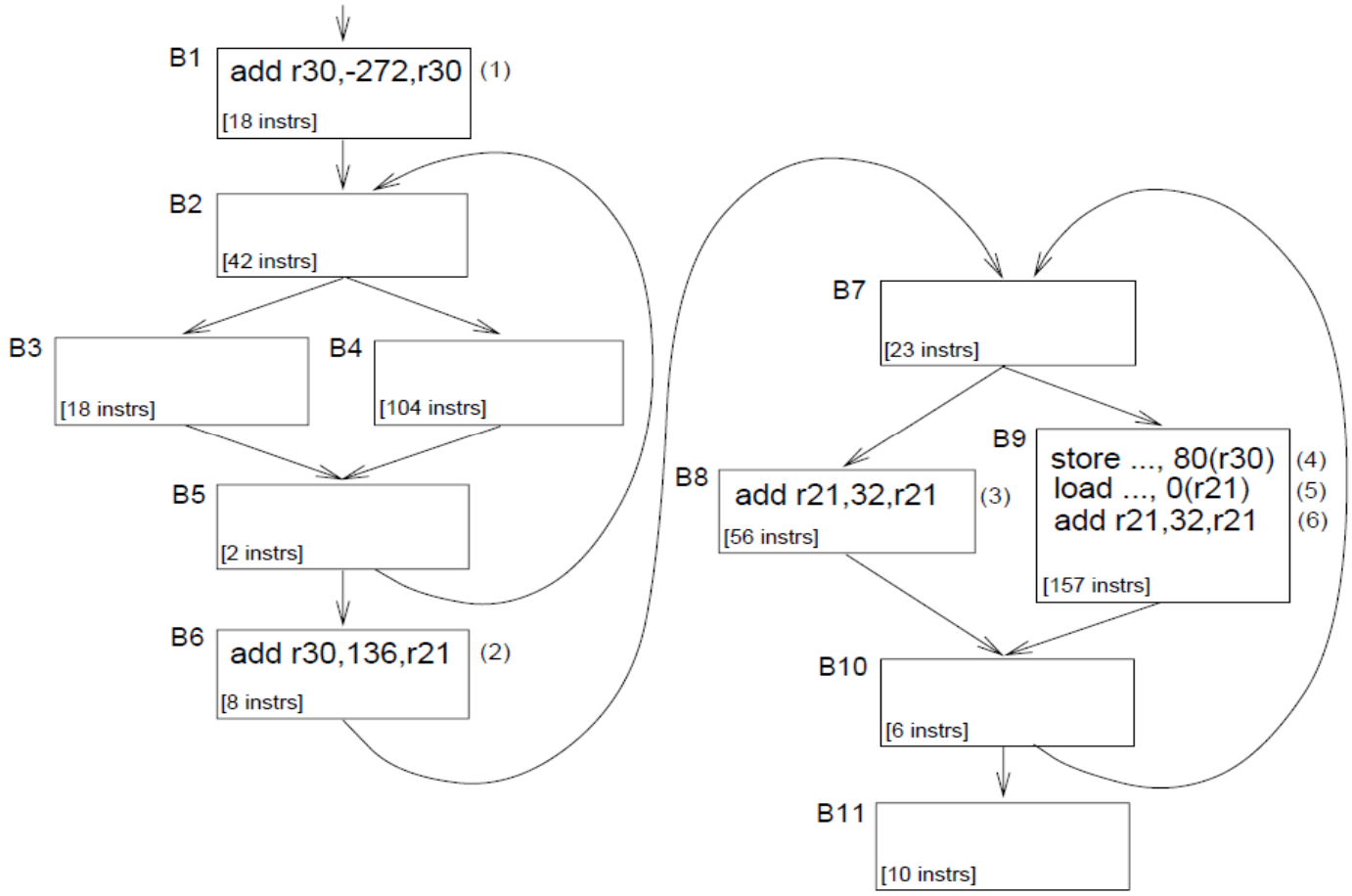


Figure 2: Flowgraph for Example 3.2 [Program: `ijpeg`; function: `jpeg_idct_ifast()`]

Register r30 is the stack pointer, r21 is used to walk through a local array of structures with a stride of 32 bytes

Example

- 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 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}>. So it can be inferred that the store instruction (4) refers to a different location than instruction (5)

Experimental Results

- The analysis was evaluated on the SPEC-95 benchmarks and agrep, a pattern matching utility, appbt, appsp, latex and nucleic2

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

Critique

- The analysis does not track the contents of memory locations
- The widening operation 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
- The context-sensitivity of the algorithm can impair the precisions
- The paper does not mention how control flow graph is constructed