

1 Free Variable Analysis

The C code, produced by our code generator, uses array copy semantics to ensure side-effect free function but also creates too many un-needed array copies. Although the copy eliminator analysis can reduce the number of array copies, the C code still has memory leak problem and requires memory de-allocation mechanism. The intuition is to find out each allocated array, which has the ownership, and safely "free" its memory space when it goes out of scope. Based on the static copy analysis in Matlab JIT compiler[4], we develop an algorithm and combine the ownership of Rust programming language[3] with the block scope in C code[7].

Free variable analysis is a forward analysis and infers the ownership and scope of each array variable. To improve the efficiency, we analyze the statements on a block, and restrict the block scope to three kinds: *function*, *while-loop* and *if-else*.

Suppose the code fragment cf contains one or more than one statements $S_b = s_0, \dots s_n$. We denote $in[cf]$ and $out[cf]$ as the set of array reference variables that has the ownership before and after cf respectively.

$$in[cf] = \cup_{cf_p \text{ is a predecessor fragment of } cf} out[cf_p]$$

$$out[cf] = gen[cf] \cup (in[cf] - kill[cf] - transfer[cf])$$

where $gen[cf]$ set generates array reference variables at cf ; $transfer$ set transfers out the array ownership at cf ; $kill$ set redefines array variables at cf .

ID	Statement	gen	$transfer$	$kill$	Description
1.a	$int[]x = [n; length]$	x	\emptyset	\emptyset	Create a new array with initial value n and array size $length$
1.b	$x = [n; length]$	x	\emptyset	x	Redefine array x with a new array
2.a	$int[]x = clone(v)$	x	\emptyset	\emptyset	Creates a copy of array v and assign the copied array to x
2.b	$x = clone(v)$	x	\emptyset	x	Redefine array x with the copy of array v
3.a	$int[]x = v$	x	v	\emptyset	Transfer the ownership from array v to x
3.b	$x = v$	x	v	x	Redefine array x and transfer ownership.

Table 1: Lists of Assignments

Function Call		Input	
		Read-only	Read-Write
Output	Return Input array	$x=f(v)$	$x=f(\text{clone}(v))$
		$gen = \{x\}$ $transfer = \{v\}$ $kill = \{\emptyset\}$	$gen = \{x\}$ $transfer = \{\emptyset\}$ $kill = \{x\}$
		$x=f(v)$	$x=f(\text{clone}(v))$
	Return New array	$gen = \{x\}$ $transfer = \{\emptyset\}$ $kill = \{\emptyset\}$	$gen = \{x\}$ $transfer = \{\emptyset\}$ $kill = \{\emptyset\}$

Table 2: Lists of Function Calls

Table 1 lists the assignments, where x and v denote the array reference variables, and also construct a table of gen , $transfer$ and $kill$ set. Table 2 shows a table of function with various input and output types, and their set

results.

Data: Function f

Result: Add "free" statement to release block-scope variables $block_vars$

```

1 Let INPUTS be the input parameters of function  $f$ ;
2 Let CFs be a list of code fragments from function  $f$ ;
3 Let BTree  $\leftarrow$  buildBlockTree( $CFs$ ) be a block-level tree, where  $CFs$ 
   are scoped into a block, and blocks are clustered into a tree structure;
4 computeBlockVars ( $BTree.root$ );
1 Procedure computeBlockVars( $b$ )
2    $outside\_vars[b] := \mathbf{INPUTS} \cup$   $p$  is a parent of  $b$   $block\_vars[p]$  ;
3    $block\_vars[b] := outside\_vars[b]$ ;
4   foreach  $cf$  in  $b$  other than ending block end do
5      $in[cf] := \cup_{cf_p \text{ is a predecessor of } cf} out[cf_p]$ ;
6      $out[cf] := gen[cf] \cup (in[cf] - kill[cf] - transfer[cf])$ ;
7     foreach  $var$  in  $kill[cf]$  do
8       if  $var \in in[cf]$  then
9         freeVar( $var$ );
10      end
11    end
12  end
13   $block\_vars[b] := in[end] - outside\_vars[b]$ ;
14  foreach  $var$  in  $block\_vars[b]$  do
15    freeVar( $var$ );
16  end
17  foreach child block  $cb$  in  $b$  do
18    computeBlockVars( $cb$ );
19  end

```

Algorithm 1: Computing Block Variables

Algorithm 1 show how we compute the set of array ownership variables, and add "free" statements to release the memory of redefined variables at each code fragment, and that of block variables at each ending block.

1.1 Challenging Memory Leaks

For some cases, our algorithm cannot make sure whether or not the variable owns the array. Thus, we need to use runtime ownership flag to keep track of the ownership change and then check the flag before freeing the variable. [6]

Listing 1: Scope Re-assignment

```

function f(i) {
  int [] x = ...
  int [] y = ... // The undecidable array variable
  boolean y_has_ownership = true; // Add runtime ownership flag

  // The ownership of 'y' is uncertain.
  if (i > 10) {
    x = [...]
    // y_has_ownership = true;
  } else {

```

```

    free(x);
    x = y; // Transfer the ownership of y
    y_has_ownership = false;
}
...
free(x);
// Depend on 'y_has_ownership' to decide whether to free y variable.
if (y_has_ownership) {
    free(y);
}
return something;
}

```

To compute the variables that require runtime ownership flag, we need to use the intersection operator to differ the outs of if-block and that of else-block variables. In this test case, block variable of if-block is x, y and block variables of else-block is x. Thus, we conclude y variable is undecidable ownership variable, and add its runtime flag to the generated code.

Listing 2: Compound Structure

```

type Board is {
    ...,
    int[] pieces // An array in Board structure
}
Board b = ...
Board p = ..
..
b.pieces = p.pieces // Clone p.pieces array and assign it to b.pieces
...
free(b); // Cannot free 'b.pieces' array.

```

This program assigns the member field (pieces), which will still be allowed at the Whitley level, but it will generate code that creates a copy of that array, rather than aliasing the array between the two data structures.

So the copy elimination analysis will reason about each whole data structure, and will sometimes allow two variables to refer to the same data structure, but will not have to worry about aliasing *within* data structures.

Listing 3: Missing Return Value

```

func(v); // Not assigning return value to a variable.

```

Due to single assignment at WYIL code level, the function call is always assigned to a register, and thus this memory leak can be avoided.

1.2 Performance Evaluation

In this section, we evaluate the impact of optimization (copy reduction and leak optimization) on 4 test cases: merge sort test cases. By using Valgrind tool, we detect memory leaks of generated C code before and after the optimization. Also, we measure the speedups.

We first generated the C code for bubble and merge sort Whitley programs without any optimization, denoted as *Naive*. Then we applied copy elimination optimization and leak optimization onto the naive C code, referred here to *Copy Opt* and *Leak Opt* respectively.

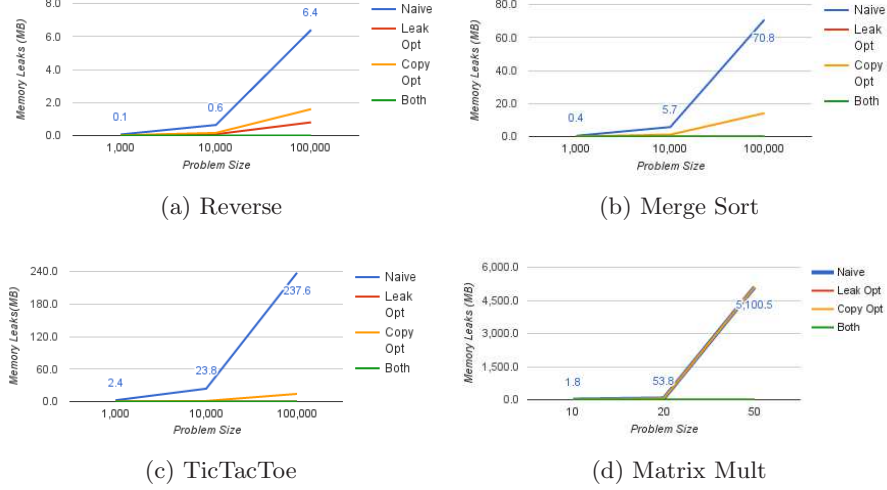


Figure 1: Memory Leaks

Figure 1 shows the memory leaks of each test case, and on our benchmark set the leak optimization can effectively reduce the memory leaks of *naive C* code down to 0 or few bytes in all test cases. The copy reduction also reduces the memory leaks but does not decrease any leak in MatrixMult test case.

Figure 2 shows the speedups of naive and optimized C code. For our benchmark set, the C code runs much faster than Java code, and scales up the performance with problem size in 'Reverse' and 'Merge Sort' test case. However, the generated C code does not gain good speedups over TicTacToe and MatrixMult programs.

2 Copy Elimination Analysis

Whiley is a functional programming language: each array needs to be copied first and then passed to the function. But too many copies lead to poor performance of implementation. The copy elimination analysis aims to reduce the number of array copies and produce efficient C code.

The intuition is to develop a straightforward analysis tool, similar to alias annotation analysis in Java[2]. The abstraction-based method[5] could give promising results over some problems but its difficulties in implementation limit its use in practice. Our analyzer works at an intermediate level of Whiley code to detect where and what array copies are un-needed using live variable analysis[1] from the Whiley compiler.

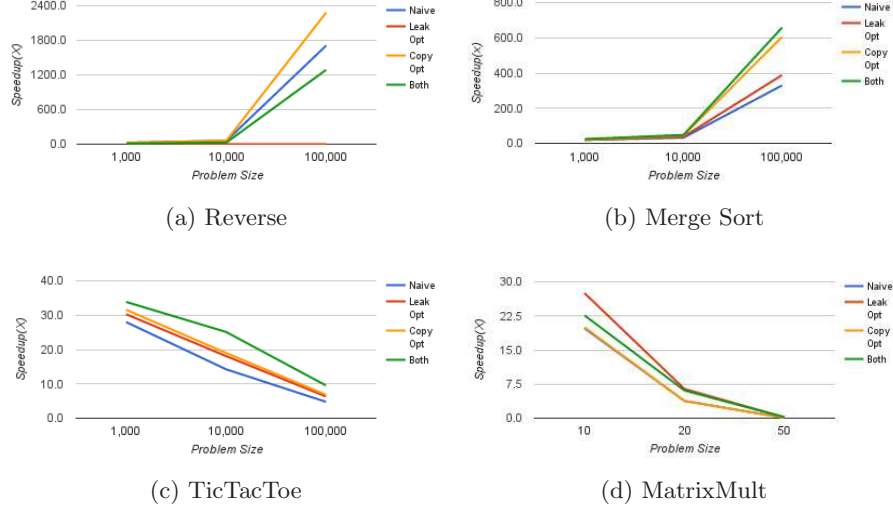


Figure 2: Speedups

Notation	Description
$in[b]$	The set of live variables before block b .
$def[b]$	The set of variables defined in block b .
$use[b]$	The set of variables used in block b .
$out[b]$	The set of live variables after block b .
B	The set of all blocks.
B^f	The set of basic blocks in function f $B^f = \{b \mid b \in B \cap F(b) = f\}$ where $F(b)$ gets the function that block b belongs to.
B_i^f	The i_{th} basic block of B^f set.
$B(c, f)$	Get the block that contains the code c of function f $B(c, f) = \{b \mid b \in B^f \cap c \in b.bytecodes()\}$

Data: Control Flow Graph of Function f
Result: $IN[B]$ and $OUT[B]$ live variable sets

```

1 for each block  $B$  other than  $EXIT$  do
2   |  $OUT[B] = \emptyset$ ;
3 end
4  $IN[EXIT] = \text{return value}$ ;
5  $OUT[EXIT] = \text{return value}$ ;
6 while any changes to  $IN$  do
7   | foreach Basic block  $B$  in reverse order other than  $EXIT$  do
8     |  $OUT[B] = \bigcup_{S \in succ[B]} IN[S]$ ;
9     |  $IN[B] = use[B] \cup (Out[B] - def[B])$ ;
10  | end
11 end

```

Algorithm 2: Live Variable Analysis

Data: Array r at Program Point p
Result: Decides whether to remove the copy of r or not

```

1  $isReadOnly = false$ ;
2 if  $p$  is a function call then
3   |  $isReadOnly = mutate(r, p)$ ;
4 end
5  $isALive = false$ ;
6 if  $p$  is an assignment then
7   |  $isALive = live(r, p)$ ;
8 end
9 if  $isReadOnly \parallel !isALive$  then
10  |  $removeCopy(r, p)$ ;
11 end
12 Procedure  $mutate(r, p)$ 
13   | Let  $f \leftarrow getFunction(p)$ ;
14   | for each program point  $p$  in  $f$  do
15     | if  $r$  is updated then
16       | return false  $\triangleright // r$  is not read-only;
17     | end
18   | end
19   | return true  $\triangleright // r$  is read-only;
20 Procedure  $live(r, p)$ 
21   | Let  $B \leftarrow getBlock(p)$ ;
22   | if  $r \in Out[B]$  then
23     | return true  $\triangleright // r$  is live after program point  $p$ ;
24   | end
25   | return false  $\triangleright // r$  becomes dead at program point  $p$ ;

```

Algorithm 3: Eliminate Array Copy

The copy analyzer first builds up control flow graph and performs live vari-

able analysis on given function, and then uses live variable analysis results to determine whether to have the array copy or in-place array update. If the array is not live or read-only, then the array copy is un-needed and can be eliminated. Consider the swap Whiley program.

```
// In={ys}
int [] xs=[0,1,2,5,4,3] // use={}, def={xs}
// Out={xs, ys}, In={xs}
ys = swap(xs, 3, 5) // use={xs}, def={ys}
// Out={xs, ys}, In={xs, ys}
assert xs == [0,1,2,5,4,3] // use={xs}, def={}
// Out={ys}, In={ys}
assert ys == [0,1,2,3,4,5] // use={ys}, def={}
// Out={}
```

The array *xs* is live at line number 4 before calling swap (or inside swap), so that array copy of *xs* is needed.

```
// In={}
int [] xs=[0,1,2,5,4,3] // use={}, def={xs}
// Out={xs}, In={}
int [] tmp = swap(xs, 3, 5) // use={xs}, def={xs}
// Out={xs}, In={tmp}
xs = tmp // use={tmp}, def={xs}
// Out={xs}, In={xs}
assert xs == [0,1,2,3,4,5] // use={xs}, def={}
```

The array *xs* is defined at line number 4, so array copies of *xs* is un-needed

References

- [1] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. *Compilers, Principles, Techniques*, chapter 9, page 608. Addison wesley, 2006.
- [2] Jonathan Aldrich, Valentin Kostadinov, and Craig Chambers. Alias annotations for program understanding. In *ACM SIGPLAN Notices*, volume 37, pages 311–330. ACM, 2002.
- [3] Rust Project Developers. Ownership, 2015.
- [4] Nurudeen Lameed and Laurie Hendren. Staged static techniques to efficiently implement array copy semantics in a matlab jit compiler. In *Compiler Construction*, pages 22–41. Springer, 2011.
- [5] Peter Schnorf, Mahadevan Ganapathi, and John L Hennessy. Compile-time copy elimination. *Software: Practice and Experience*, 23(11):1175–1200, 1993.
- [6] Manish Virmani. Pointers and memory leaks in c.
- [7] Wikipedia. Block scope — Wikipedia, the free encyclopedia, 2015. [Online; accessed 11-October-2015].