Memory Optimization for C implementations of Whiley

Min-Hsien Weng[1] Bernhard Pfahringer[2] Mark Utting[3]

[1]Computer Science Department
Waikato University

[2]Computer Science Department
Waikato University

[3]School of Business
University of the Sunshine Coast

SAPLING16

Whiley

Whiley[1] is a new programming language designed to:

- Provide ease-to-use syntax (e.g. Python),
- Verify a program with given specifications, and detects runtime errors at compile time, and
- Deploy Whiley programs to existing systems (e.g. Whiley to Java code)

The use of value semantics in Whiley makes program verification easier, but poses a potential threat to program efficiency.

[1] Pearce, David J., and Lindsay Groves. "Designing a verifying compiler: Lessons learned from developing Whiley." Science of Computer Programming 113 (2015): 191-220.

Whiley to C

- We develop C code generator for Whiley.
- Call-by-Value semantics causes our naive C code, translated from Whiley, to have several performance issues:
  - Excessive copying overheads as all arrays are copied before each modification
  - Severe memory leaks as all arrays are allocated on the heap, and not de-allocated
- We apply static analysis techniques to improve the efficiency
  - Bound analysis[2] finds appropriate integer types
  - Copy analysis eliminates unnecessary copies
  - De-allocation analysis avoids most of memory leaks

[2]Weng, Min-Hsien, Mark Utting, and Bernhard Pfahringer. "Bound Analysis for Whiley Programs." Electronic Notes in Theoretical Computer Science 320 (2016): 53-67.

Copy Elimination Analysis

Call-by-value code makes a copy for every value

Examples

a = copy(b)   a = foo(copy(b))

Those copies are not needed when

- b becomes dead afterwards, or
- b is passed as read-only parameter

Memory Deallocation Analysis

- Copy analysis introduces memory aliasing and makes it hard to find the right variable to free the allocated memory space.
- Every array variable (a) is associated with a runtime flag ($a\_dealloc$), to indicate if this variable is responsible for de-allocation.
- Our deallocation analysis needs to preserve the invariant:

Theorem (De-allocation Invariant)

At any program point, exactly one variable is responsible to free the allocated memory space. Note environment e maps a variable to its value.

$$
\begin{aligned}
&(a_{dealloc} \wedge e(a) \neq NULL) \\
&\Rightarrow (\forall var : VARS \bullet (var \neq a \wedge e(var) == e(a)) \\
&\qquad \Rightarrow e(var_{dealloc}) = false)
\end{aligned}
$$

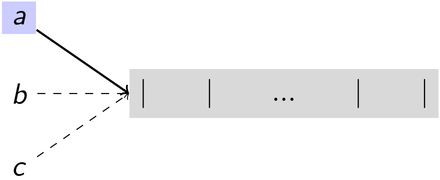
Memory Deallocation Analysis

Free an array variable using *PRE_DEALLOC* macro before each update AND at the end of its scope.

```
/* De-allocate variables before exit,
to avoid memory leaks */
PRE_DEALLOC(a);
PRE_DEALLOC(b);
PRE_DEALLOC(c);

return 0;
```

$e(a\_dealloc) = true$
$e(b\_dealloc) = false$
$e(c\_dealloc) = false$



$PRE\_DEALLOC(a)$

expands to:

```
/* Check the flag to free,
 or not to free a */
if(a_dealloc){
 free(a);
 a=NULL;
 a_dealloc=false;
}
```

Memory Deallocation Analysis

For a function call $a := foo(b)$, our analysis bases on below results to

- Copy or not copy b, and
- Choose a macro[3] to specify caller/callee to free the passing b
  $a := foo(b,\ b\_dealloc)$

Function call $a := foo(b)$					
foo Mutates b?		F	F	T('may-be')	T('may-be')
foo Returns b?		F	T('may-be')	T('may-be')	F
b is live at caller?	F	No Copy RETAIN	No Copy RESET	No Copy RESET	No Copy RETAIN
	T ('may-be')	No Copy RETAIN	No Copy RESET	Copy CALLER	Copy CALLEE

[3] RETAIN, RESET, CALLER and CALLEE macros can be expanded to C code to change/maintain flag values and specify the de-allocator for the passing parameter

Reverse Example — Copy Elimination

```
// Reverse an array (Callee)
int[] reverse(int[] arr){
  ...
  int[] r = malloc(...);
  while(i > 0){
    int item = arr[|arr|-i];
    i = i - 1;
    r[i] = item;
  }
  return r;
}
// Main entry (Caller)
void main(int argc, ...){
  ...
  //Read-only 'arr'
  tmp = reverse(copy(arr));
  //Assertion
  assert arr[0] == tmp[3];

  //Temporary variable
  out = copy(tmp);

  return 0;
}
```

Remove un-necessary copies:

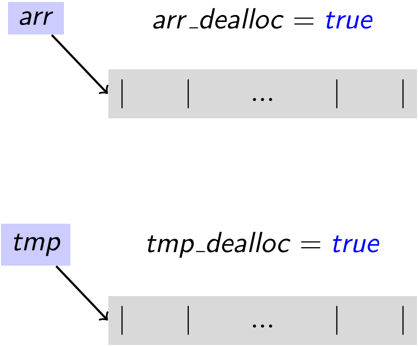
- Copy of *arr*
  - *arr* is NOT returned by *reverse*
  - *arr* is NOT written by *reverse*
  - *arr* is still alive at *main*

  Pass read-only *arr* to reverse function
- Copy of *tmp*

Reverse Example — Function Call

```
void main(...){
 ...
 PRE_DEALLOC(tmp);
 /* Do not free the shared
    array at reverse */
 tmp = reverse(arr, false);

 RETAIN(tmp, arr);
 ...
}
```



$RETAIN(tmp, arr)$

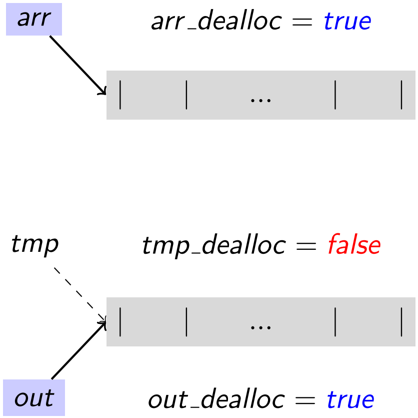
expands to:

```
// No changes to arr flag
tmp_dealloc = true;
```

Reverse Example — Assignment

```c
void main(...){
 ...
 PRE_DEALLOC(tmp);
 // Alias out to tmp
 out = tmp;

 TRANSFER(tmp, out);
 ...
}
```



$TRANSFER(tmp, out)$

expands to:

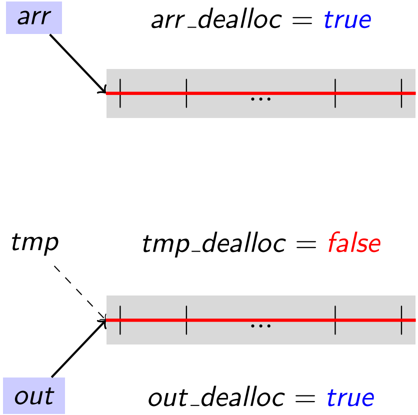
```c
out_dealloc = tmp_dealloc;
tmp_dealloc = false;
```

TRANSFER macro is similar to move semantics in Rust

Reverse Example — Return

```
void main(...){
 ...
 // Free un-used variables
 PRE_DEALLOC(tmp);
 PRE_DEALLOC(arr);
 PRE_DEALLOC(out);

 return 0;
}
```

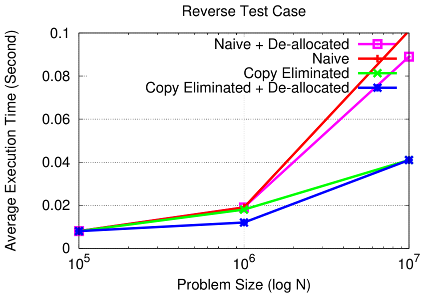


Benchmarks

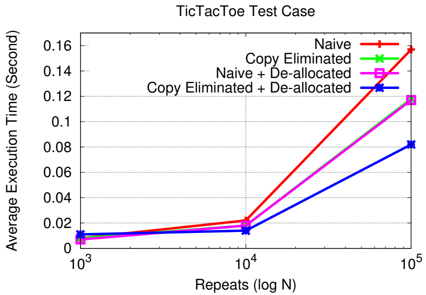
- Benchmark suite includes Reverse, TicTacToe, Bubblesort, Mergesort and MatrixMult examples
- Each benchmark program is translated into 4 kinds of C code:
  - Naive: no optimization (Naive)
  - Naive + Deallocation: de-allocation analysis only (N+D)
  - Copy Eliminated: copy analysis only (C)
  - Copy Eliminated + Deallocation: both copy and de-allocation analysis (C+D)
- Performance evaluation
  - Average execution time (GCC 5.4.1)
  - Memory leaks using Valgrind (v.3.10.1)

Average Execution Time — C Code

Reverse Example:



TicTacToe Example:

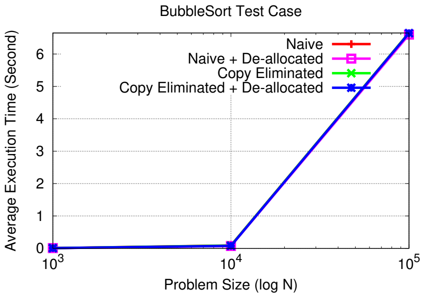


Speedup(C vs.Naive) = 2.46x
Speedup(C+D vs. Naive) = 2.49x

Speedup(C vs.Naive) = 1.3x
Speedup(C+D vs. Naive) = 1.9x

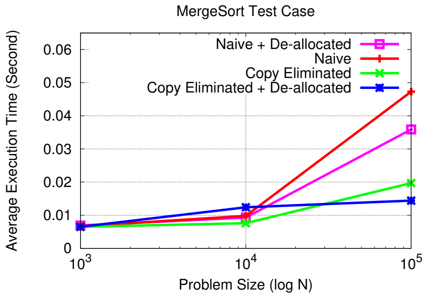
Average Execution Time

Bubble Sort Example:



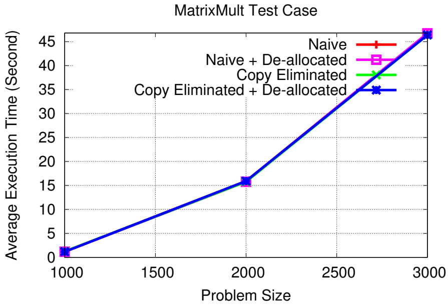
No speedups

Merge Sort Example:



Speedup(C vs.Naive) = 2.4x
Speedup(C+D vs. Naive) = 3.3x

Average Execution Time



MatrixMult Example:
Profiling (gprof) execution time

- 99% time on calculating dot products of row and col (Possible parallelism?)
- 1% time on copying matrix array

Computation dominates copy overheads

Memory Leaks (MB)

Our analysis effectively avoids all memory leaks of 5 examples.

	N	N + D	C	C + D
Reverse				
(100,000)	4.8	0	1.6	0
(1,000,000)	48	0	16	0
(10,000,000)	480	0	160	0
TicTacToe				
(1,000)	2.7	0	2.0	0
(10,000)	27	0	20.4	0
(100,000)	276	0	204	0

	N	N + D	C	C + D
BubbleSort				
(1,000)	0.03	0	0.008	0
(10,000)	0.3	0	0.08	0
(100,000)	3.2	0	0.8	0
MergeSort				
(1,000)	0.35	0	0.08	0
(10,000)	4.6	0	1.14	0
(100,000)	56	0	14.1	0
MatrixMult				
(1,000)	152	0	24	0
(2,000)	608	0	96	0
(3,000)	1.36GB	0	216	0

Memory leaks of naive C code increase with problem size, and would exhaust all available memory (e.g. 12,000 matrix size uses up 16GB and stops the program).

Related Work

- Copy elimination
  - Reference counting (GC) at runtime requires extra overheads, particularly on multi-threads
  - Static analysis (MATLAB compiler) at compile-time, similar to ours.
- Memory deallocation
  - Rust single ownership rule is validated by move semantics at compile-time
    - Every value has a single owner at any given time, similar to ours
    - But we keep track of the deallocation responsibility dynamically
  - C++11 smart pointers can be deleted automatically by runtime

Conclusion

- Copy analysis reduces un-necessary copies and gives good speed-ups.
- De-allocation analysis drops off unused arrays at an appropriate time
  - Chooses macros to change runtime deallocation flag value, and ensures single deallocation invariant
  - Prioritizes memory safety, but still can have unavoidable memory leaks
    - ⋆ Mutually recursive function calls
    - ⋆ Uncertain function behaviours (may-be return or may be read-write) causes extra copies and has memory leaks at callee

If you require any further information, feel free to contact me (mw169@students.waikato.ac.nz).