Static Techniques for Reducing Memory Usage in the C Implementation of Whiley Programs

Min-Hsien Weng
Computer Science Department
University of Waikato
Hamilton, New Zealand
mw169@students.waikato.ac.nz

Bernhard Pfahringer
Computer Science Department
University of Waikato
Hamilton, New Zealand
bernhard@waikato.ac.nz

Mark Utting
School of Business
University of the Sunshine Coast
Queensland, Australia
utting@usc.edu.au

ABSTRACT

Languages that use call-by-value semantics, such as Whiley, can make program verification easier. But efficient implementation becomes harder, due to the overhead of copying and garbage collection. This paper describes how a mixture of static analysis and runtime-monitoring can be used to eliminate unnecessary copying and deallocate memory without garbage collection. We show that this allows Whiley programs to be translated into efficient C implementations.

CCS Concepts

•Theory of computation → Program analysis; •Software and its engineering → *Correctness; Software performance;*

Keywords

Static Analysis; Copy Elimination; Compiler;

1. INTRODUCTION

Whiley[8] is a hybrid object-oriented and functional programming language that uses extended static checking to eliminate errors at compile time. To make automated reasoning about programs easier, it uses unbounded integers, requires call-by-value semantics for all data structures, and limits aliasing. It has a clear separation between functions and methods, so that functions cannot have side-effects and aliasing is not possible within functions.

To enable Whiley to be used to write efficient programs, particularly for embedded systems with limited memory [7], we have developed a Whiley-to-C translator. However, a naive translation of Whiley into C has several significant efficiency problems: a) it copies arrays and structures each time they are assigned, which is slow and memory-hungry; b) it lacks garbage collection, so leads to memory leaks in the C code, which means that programs cannot run for long.

This paper describes several static code analysers that we have developed to eliminate unnecessary copying of memory, as well as memory deallocation strategies that use a mixture of static analysis and dynamic tracking to ensure that each memory block is deallocated exactly once, in order to avoid memory leaks. With these optimizations, our code generator can produce efficient sequential C code while retaining program safety.

The Whiley compilation system[8] is based around the Whiley Intermediate Language (WyIL), which uses custom bytecodes to represent programs in a structured way. The default code generator generates unoptimised Java source code and relies on garbage collection and BigInteger libraries for all computations, which results in very slow execution times.

Our C code generator takes WyIL as input, iterates through all function blocks and translates each WyIL code into sequential C99-compatible C code. It chooses a fixed-size integer type for each variable, which in this paper is the C 'long long' type - in general the code generator uses static bounds analysis to infer integer bounds and choose appropriate integer types for variables, but that is outside the scope of this paper.

Array and structure data values are heap-allocated, so require explicit deallocation, to avoid memory leaks and double freeing problems. The language is constrained so that cycles in data structures are not allowed. While it would be possible to use reference counting to deallocate memory, we want to avoid the runtime overheads of reference counting, to enable efficient execution on small embedded computers.

This paper describes how we use static analysis of the program to eliminate unnecessary memory allocation and copying, and a mixture of static analysis and runtime flags to determine a unique point where each allocated memory block can be deallocated. These analysis techniques are not guaranteed to eliminate all unnecessary copying, or to handle all input Whiley programs. If they fail to find a safe strategy, our C code generator reports an error and we require the programmer to modify the input Whiley program so that it fits within the subset of the language that can be safely translated.

Section 2 of the paper describes the various static analyses and code generation strategies that we use. Section 3 gives a performance evaluation of our techniques on five case studies. Section 4 discusses related work, and Section 5 discusses our conclusions.

2. CODE ANALYSIS

Our project develops several function analyses, copy elimi-

Accepted for publication in the Proceedings of the Australasian Computer Science Week 2017, ACM Press

nation analysis and deallocation analysis to extract the properties of each WyIL code, and then assist our code generator to apply code optimization and produce efficient code.

Our analysis allows simple non-recursive, and directly-recursive function calls only. Other mutual, anonymous or complicated recursions, which may cause infinite loops, are not supported, so that the termination of our analysis can be expected.

2.1 Function Analyses

The function analysers all employ a conservative strategy to extract variable information from functions, and stores that information in order to support the copy and deallocation analysers to make safe code optimization, while improving the efficiency.

All function analysis builds up a *call graph* prior to the analysis. The construction of a call graph starts from the 'main' function to go through each function call and explore deeper calls recursively if needed, to create a tree structure with a set of caller and callee (nodes), and calling relation (edges). For a recursive function call, the exploration adds one branch and does not go into the call, as simple recursion is assumed in our project.

The function analyser then traverses each function in the post order of call graph and processes specific information. The bottom-up approach ensures when analysing a function call, the analyser can obtain the pre-optimized results of the functions that are being called, rather than raw and un-optimized results. Our project includes three kinds of function analysers:

- The read-write analyser checks if a variable is or may be read and written inside a function
- The return analyser checks if a variable is or may be returned by a function.
- The live analyser checks if a variable is alive or used after the code of a function.

2.1.1 Read-Write Analyser

Algorithm 1 Read-Write Analysis

Input: Call Graph
Output: MUT maps each function to a read-write set

```
1: MUT = ∅
2: for each f function in post order of call graph do
3:    MUT(f) = ∅
4:    for each code in f do
5:       lhs ← Extract LHS variable at code
6:       MUT(f) = MUT(f) ∪ lhs
7:    end for
8: end for
```

The left-hand side (LHS) variable is used to store the computation result of a code, so is considered to be a mutable or read-write variable and added to the set (see Algorithm 1). The variable at the right-hand side (RHS) is usually not mutable, because it is copied before update. However, if our copy-elimination causes it to become aliased with the mutable variable then it can also appear in the result set of the read-write analyser.

Algorithm 2 Mutable Check

Input: v variable, f function
Output: Return true if r is mutated inside f function
return $v \in MUT(f)$

Our read-write analysis conservatively keeps all 'definite' and 'may-be' mutable variables. The mutable check (see Algorithm 2) only weakly identifies a mutable variable, but can strongly detect an immutable or read-only variable. This information about read-only variables is used by the copy analyser to decide whether copying is necessary or not.

2.1.2 Return Analysis

Algorithm 3 Return Analysis

Input: Call graph
Output: RET maps each function to a return set

```
1: RET = ∅
2: for each f function in post order of call graph do
3:    RET(f) = ∅
4:    for each code in f do
5:       if code is Return then
6:          ret ← Extract return variable from code
7:          RET(f) = RET(f) ∪ ret
8:       end if
9:    end for
10: end for
```

The return analyser (see Algorithm 3) includes all definite and possible return variables, even those inside if-else. The return information allows the update from the copy analyser to add 'may-be' or aliased return variable after copy removal.

Algorithm 4 Return Check

Input: v variable, f function
Output: Return true if v is returned by f function
return $v \in RET(f)$

Due to the expansion of the return set, the return check (see Algorithm 4) can be used to effectively detect those non-returnable variables that are never returned by the function, which can allow that memory to be deallocated within the function. As opposed to strong definitely-returned results, this check may mistakenly report a variable as returnable, when it is not actually returned, and skip the memory deallocation. Despite the potential memory leaks, the conservative false alarm can reduce the chances of invalid freeing while retaining memory safety.

2.1.3 Live Variable Analysis

Live variable analysis[1] is widely applied in compilers to determine whether a variable is still live (used) after a code, using an iterative backwards data-flow algorithm.

The live variable analyser first builds up the *control flow graph* of a function. Then it scans each WyIL code, and based on the code type, constructs the corresponding block to store the code. For example, for a loop the analyser creates a loop structure and assigns each loop code to loop header, loop body and loop exit respectively. Other block types include entry and exit blocks, if-else branches, and normal blocks.

Notation	Description
$in(b,\ f)$	The set of live variables before b block of f function
$def(b,\ f)$	The set of variables defined in b block of f function
$use(b,\ f)$	The set of variables used in b block of f function
$out(b,\ f)$	The set of live variables after b block of f function

Table 1: Notations of Live Variable Analysis

Algorithm 5 Live Variable Analysis

Input: Call graph
Output: *out* maps (block,function) to its live variables

for each f function in post order of call graph **do**
  $CFG \leftarrow$ build control flow graph of f function
  $blocks \leftarrow$ Traverse post-order blocks of CFG
  for each b block in $blocks$ other than exit **do**
    $out(b,\ f) = \emptyset$
  end for
  //Special in and out set for exit block
  $in(exit,\ f) = \{return\ variable\}$
  $out(exit,\ f) = \{return\ variable\}$
  //Search live variables until fixed-point
  while Changes to any $in(b,\ f)$ set **do**
    for each b block in $blocks$ other than exit **do**
      $out(b,\ f) = \bigcup_{S \in succ[b]} in(S,\ f)$
      $in(b,\ f) = use(b,\ f) \cup [out(b,\ f) - def(b,\ f)]$
    end for
  end while
end for

Secondly, the live variable analyser backtracks through each block and computes the liveness equation to find the set of live variables before and after a block repeatedly, until the input sets reach a fixed point. This iterative procedure can discover the live variables inside a loop block and if-else branch, but ensures a comprehensive result set.

Thirdly, the analyser separately processes the 'exit' block, because the return variable must be alive both before and after the block. Finally, the analyser follows the above steps to find the liveness information for all functions.

Algorithm 6 Liveness Check

Input: v variable, *code*, f function
Output: Returns true if v is alive at *code* in f function

$b \leftarrow$ Locate the block that contains *code* in f function
return $v \in out(b,\ f)$

The set of live variables can determine if a variable is still used after a specific code (see Algorithm 6)

2.2 Copy Elimination Analysis

By default, Whiley uses copy semantics for every value. For example, an assignment makes a copy of the right-hand side variable ($int[]\ a = copy(b)$), and function call copies the passing parameter ($int[]\ a = func(copy(b))$). But these copies are unnecessary when:

- b is dead (not used) after the code, or
- b is passed to a read-only function parameter.

Algorithm 7 Copy Elimination Check

Input: v variable, *code*, f function
Output: Return true if the copy of v can be eliminated

$isLive \leftarrow$ check if v is alive at *code* of f
if v is NOT alive **then**
  return true //Copy can be removed
end if
if code is a function call **then**
  //Special check for passing parameter v
  $v_{callee} \leftarrow$ map v to local variable at callee
  if v_{callee} is NOT mutated at *callee* **then**
    return true //Copy can be removed
  end if
end if
return false //Copy is needed in remaining cases

The copy analyser detects what copies can be eliminated using backward live variable analysis along with a decision procedure (see Algorithm 7). The procedure removes the copying of dead variables, which are not used afterwards, as well as avoiding copying structures that are passed to read-only function parameters. Other structure copying commands are conservatively kept to avoid misuse of memory aliases.

Algorithm 8 Copy Elimination Analysis

Input: Call graph
Output: Generated C code

Initialize function analysers to perform read-write analysis, return analysis and live variable analysis.
for each f function in post order of call graph **do**
  for each *code* in f **do**
    for each *var* in right-hand-side of code **do**
      isCopyEliminated
      = `COPY_ElIMINATION_CHECK(var, code, f)`
      if $isCopyEliminated = true$ **then**
        Generate the copy-reduced code
        Update the read-write and return sets with *var*
      else
        Generate the naive code
      end if
    end for
  end for
end for

The copy elimination analysis (see Algorithm 8) iterates through each code, checks if the copy of each right-hand side variable can be removed using the decision procedure, and then passes the resulting flag to the code generator, which produces the corresponding C code.

Once the copy is removed, the variable may become aliased to an existing read-write or return variable. To account for this, the read-write and return sets of the current function are updated, to ensure the copy analyser obtains consistent and optimized results for all functions.

2.3 Deallocation Analysis

Arrays and structures are declared as pointers, and explicitly allocated and deallocated as heap memory in the generated C code. As a result, incorrect memory management

could cause critical memory safety problems, e.g. memory leaks or double freeing.

Intuitively, a variable that is no longer used, but is still bound to a previously allocated memory structure, must be deallocated. As a result of copy-elimination, our generated C code does sometimes have two variables pointing to the same memory location, so to ensure that every memory structure is deallocated exactly once, we associate a boolean *deallocation flag* with each heap variable to indicate whether that variable is responsible for deallocating the memory. In the generated C code we use *deallocation macros* to update the deallocation flags each time a heap variable is updated. A crucial invariant of the generated code is that whenever two or more heap variables are aliased to the same memory structure, then only one of those variable should have its deallocation flag set to true, indicating that it is responsible for deallocating that memory structure.

2.3.1 Deallocation Invariant

THEOREM 1. ***Deallocation Invariant*** *For every allocated structure, and before every WyIL code, there is exactly one variable that points to that structure and has the deallocation flag across all function scopes. Given an environment* `e` *that maps variable names to values, this invariant* `inv` *is defined as:*

$$\forall i, j : VARS \bullet (i \neq j \wedge e(i_{dealloc}) \wedge e(i) \neq NULL \wedge e(i) == e(j)) \Rightarrow e(j_{dealloc}) = false$$

where $VARS$ denotes the set of all variables, and $i_{dealloc}$ and $j_{dealloc}$ denote the deallocation flags of variable i and j respectively.

The general invariant can be narrowed down to a given variable, i.e. `inv(a)`

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

This deallocation invariant ensures that at any program point only one variable has the deallocation flag set to true, to free the allocated memory space. This invariant allows multiple variables to share the same allocated memory structure but restricts only one variable to be responsible for deallocating the memory structure.

The deallocation analyser takes each code as input, analyses the code properties and applies one of the following macros to update a variable and change its runtime deallocation flag. Table 2 shows which macro is used in case. The correct choice of macro ensures that the deallocation invariant is maintained before and after each line of code.

2.3.2 Deallocation Macros

The deallocation analyser takes each WyIL code as input, and adds a *pre-deallocation* macro and a *post-deallocation* macro to the generated C code, to release the old memory and make changes to the deallocation flag respectively.

Any time that the value of a heap variable is about to be overwritten, it is important to check whether it is responsible for deallocating that memory structure. Our code generator generates a call to the following *pre-deallocation* macro to do this check.

```
1 #define PRE_DEALLOC(a)
2 if(a_dealloc){
3  free(a); a:=NULL; a_dealloc:=false;
4 }
```

Pre-Deallocation Macro. The `PRE_DEALLOC` macro empties the left-hand side variable prior to a statement, to avoid any memory leak caused by the update. However, when applying to a return code, the pre-deallocation macro is applied to all variables (excluding the return variables) as well as all parameter variables, in order to reclaim unused memory before the function exits.

Post-Deallocation Macros. After each statement, one of the following *post-deallocation* macros is called to update the heap variables and make changes to the deallocation flags. According to code type and copy information, the macros are defined as follows:

Assignment. An assignment *may* or *may not* copy right-hand side variable (source) into the left-hand side variable (destination). The post-deallocation macro can be split into two cases:

```
1 #define ADD_DEALLOC(a, b)
2 PRE_DEALLOC(a);
3 a := copy(b);
4 a_dealloc := true;
```

The `ADD_DEALLOC` macro lets the destination point to a fresh copy of the source variable structure. Due to having separate memory structures, the macro sets the destination deallocation flag to true, but leaves the source deallocation flag unchanged as no change has occurred to that variable.

```
1 #define TRANSFER_DEALLOC(a, b)
2 PRE_DEALLOC(a);
3 a := b;
4 a_dealloc := b_dealloc;
5 b_dealloc := false;
```

The `TRANSFER_DEALLOC` macro aliases the source and destination to the same memory structure, so transfers the deallocation flag from the source to the destination and resets the source flag, to ensure that only the destination variable will be responsible for deallocation.

Function Call. A function call passes parameters to the calling function (callee) and then returns the result back to caller site. As a function call *may* or *may not* create a fresh copy of each passing parameter, the deallocation problem involves:

- when the parameter copy is made, should the callee or caller free the passing parameter?
- when the parameter copy is eliminated, should the callee or caller free the passing parameter?

The post-deallocation macro specifies the caller to free function return (destination), and appends one flag value along with each parameter (source) to the function call, to indicate whether the passing parameter can be freed by

Function call $a := func(b)$					
$func$ Mutates b?		F	F	T(‘may-be’)	T(‘may-be’)
$func$ Returns b?		F	T(‘may-be’)	T(‘may-be’)	F
b is live at caller?	F	No Copy RETAIN_DEALLOC	No Copy RESET_DEALLOC	No Copy RESET_DEALLOC	No Copy RETAIN_DEALLOC
	T (‘may-be’)	No Copy RETAIN_DEALLOC	No Copy RESET_DEALLOC	Copy CALLER_DEALLOC	Copy CALLEE_DEALLOC

Table 2: Post-deallocation Macro for Function Call

callee. The flag value is determined by taking account of *mutable*, *return* and *liveness* analysis as shown in Table 2. Note that these macros are induced from simulation results with all possible combinations of flag values, and validated by checking that all the test cases have no memory leaks.

Function Call of Copied Parameter

The parameter is passed to a function call with a copy as the parameter is or may be mutated by callee, but the original value is used after the function call.

```
#define CALLER_DEALLOC(a, b)
PRE_DEALLOC(a);
// Do not free copied ‘b’ at ‘func’
a := func(tmp := copy(b), false);
// Possible memory leak on copied ‘b’
if (a != tmp) {free(tmp); tmp=NULL;}
a_dealloc := true;
```

The `CALLER_DEALLOC` macro is applied when the parameter is or may be returned by the function call and avoids being freed by callee. Due to ambiguous results of return analysis, this macro would make an extra copy and lead to potential memory leaks. For example, the calling function contains an if-else to output different returns (a new array or copied b array). The ‘may-be‘ return, if it is not actually returned, skips the de-allocation of passing parameter within callee and leaves the extra copy un-deallocated after the function exits. Such memory leaks can be avoided by the additional de-allocation check.

The conservative caller macro is a trade-off between memory leaks and memory safety, to deal with the uncertainty on function return at runtime and avoid wrongly nullifying the return.

```
#define CALLEE_DEALLOC(a, b)
PRE_DEALLOC(a);
// Free copied ‘b’ at ‘func’
a := func(tmp := copy(b), true);
// No change to ‘b_dealloc’
a_dealloc := true;
```

The `CALLEE_DEALLOC` macro is applied when the passing parameter is NOT returned by function call. So the parameter can be deallocated separately at callee since it is not aliased with function return.

Function Call of Not Copied Parameter

The parameter is passed straight to a function call without copy. Due to being used and shared by caller and callee, the passing parameter, if freed within callee, may cause dangling pointers and make use of invalid data at caller site. So the de-allocation of un-copied parameter is always delegated to the caller.

```
#define RETAIN_DEALLOC(a, b)
PRE_DEALLOC(a);
// Do not free ’b’ at ’func’
a := func(b, false);
// No change to ’b_dealloc’
a_dealloc := true;
```

The `RETAIN_DEALLOC` macro is applied when the parameter is not returned by function call. Since the parameter is not aliased with function return, its flag at caller site can stay unchanged.

```
#define RESET_DEALLOC(a, b)
PRE_DEALLOC(a);
// Do not free ’b’ at ’func’
a := func(b, false);
// Take out ‘b’ flag
b_dealloc := false;
a_dealloc := true;
```

The `RESET_DEALLOC` macro is applied when the passing parameter is or may be returned by calling function and aliased with function return, so specifies the de-allocation at caller site.

3. PERFORMANCE EVALUATION

The benchmark suite includes `Reverse`, `TicTacToe`, `Merge Sort`, `Bubble Sort` and `Matrix Multi` test cases (code snippets are listed in Appendix A). Each test case is translated by our code generator, both with and without the copy-elimination and the deallocation analysers, giving four kinds of C implementations:

- Naive C code (`N`) is translated from WyIL code without any code optimization.
- Naive and memory deallocated C code (`N+D`) is translated from WyIL code with deallocation macros.
- Copy-eliminated C code (`C`) is translated from WyIL code with just copy elimination.
- Copy-eliminated and memory deallocated C code (`C+D`) is translated from WyIL code with copy elimination and deallocation macros.

All benchmarks are conducted on Ubuntu machine (i7-4770 CPU @ 3.40GHz and 16 GB memory), and compiled into executables by GCC compiler (version 5.4.1) and run with three problem sizes.

Speed-up. Each implementation is repeatedly run 10 times on each problem size and averaged the execution time to calculate the speed-up over naive code. Speed-up results in Table 3 show that the extra deallocation macro may slightly slow down the naive code in two cases. However, without deallocation macro the naive code can not be run on large problem size due to severe memory leaks. For example, when

Table 3: Speed up

Test Case	Problem Size	Average Execution Time (Seconds)				Speedup (vs. Naive code)		
		N	**N + D**	**C**	**C + D**	**N + D**	**C**	**C + D**
Reverse	100,000	0.008	0.011	0.007	0.007	0.72	1.09	1.13
	1,000,000	0.017	0.016	0.012	0.010	1.02	1.44	1.63
	10,000,000	0.090	0.099	0.043	0.043	0.91	2.11	2.12
TicTacToe	1,000	0.011	0.019	0.008	0.007	0.59	1.45	1.48
	10,000	0.023	0.018	0.019	0.015	1.22	1.18	1.51
	100,000	0.166	0.113	0.122	0.082	1.46	1.36	2.02
Bubble Sort	1,000	0.01	0.01	0.01	0.01	1.00	1.03	0.99
	10,000	0.07	0.07	0.07	0.07	1.00	1.01	0.98
	100,000	6.61	6.62	6.63	6.62	1.00	1.00	1.00
Merge Sort	1,000	0.007	0.007	0.007	0.008	1.02	0.93	0.85
	10,000	0.010	0.009	0.008	0.007	1.10	1.18	1.38
	100,000	0.043	0.061	0.017	0.015	0.70	2.59	2.94
Matrix Mult	1,000	1.30	1.23	1.27	1.19	1.06	1.02	1.09
	2,000	15.85	15.73	15.99	15.71	1.01	0.99	1.01
	3,000	46.73	46.68	46.62	46.55	1.00	1.00	1.00

matrix size is increased upto 10,000 the naive code runs out of system memory and causes system breakdown.

The combined copy elimination and de-allocation analysis can scale up the speed-up with problem size in three cases ('Reverse', 'TicTacToe' and 'Merge Sort'). For other two cases, the performance requires different optimizations. By profiling the total execution time using 'gprof' tool, the naive 'Bubble Sort' spends almost 100% time on sorting and swapping array items. Likewise, the naive 'MatrixMult' code uses 99% time to calculate the products of rows and columns, and only 0.1% time on array copying. As their computation time dominates copying overheads, our memory optimization has little improvement on the total execution time.

Table 4: Memory Leaks (Bytes)

	N	**N + D**	**C**	**C + D**	Copy Elimination(%)
Reverse					
(100,000)	4,800,256	0	1,600,248	0	66.66%
(1,000,000)	48,000,264	0	16,000,256	0	66.67%
(10,000,000)	480,000,272	0	160,000,264	0	66.67%
TicTacToe					
(1,000)	2,760,280	0	2,040,272	0	26.08%
(10,000)	27,600,288	0	20,400,280	0	26.09%
(100,000)	276,000,296	0	204,000,288	0	26.09%
Bubble Sort					
(1,000)	32,264	0	8,256	0	74.41%
(10,000)	320,272	0	80,264	0	74.94%
(100,000)	3,200,280	0	800,272	0	74.99%
Merge Sort					
(1,000)	351,488	0	88,056	0	74.95%
(10,000)	4,595,976	0	1,149,184	0	75.00%
(100,000)	56,605,968	0	14,151,688	0	75.00%
Matrix Mult					
(1,000)	152,000,784	0	24,000,624	0	84.21%
(2,000)	608,000,784	0	96,000,624	0	84.21%
(3,000)	1,368,000,784	0	216,000,624	0	84.21%

Memory Leaks. Table 4 shows the memory leaks of each generated code, detected by *Valgrind* tool [6] and summed up 4 kinds of memory leaks (definitely loss, indirectly loss, possibly loss and still reachable loss).

The results show that, on our benchmark suite, our deallocation analysis effectively avoid memory leaks in all test cases. Also, the copy elimination alone can reduce upto 84% leaks in 'MatrixMult' case. Note that apart from these 5 cases, our deallocation analysis does not guarantee 'zero' leaks as there can be potential leaks in `CALLER_DEALLOC` macro.

4. RELATED WORK

Copy Elimination. Many attempts to solve extra copies in value-semantic programs rely on dynamic reference counting technique for its prompt memory reclaim. But the overhead cost of garbage collection may slow down the program execution at run-time.

The copy elimination approach at compile time[9] was proposed to optimize the compiler and remove large amounts of copies using aggregate analysis along with a set of techniques, to produce efficient code without run-time support. For example, an alternative static analysis[5] was implemented in `MATLAB JIT` compiler to remove un-necessary copies while compiling the program even though `MATLAB` runtime has been built in with reference counting.

Our static copy analysis, similar to alias annotation analysis for Java language[2], works at intermediate level of Whiley code and uses live variable analysis[1] and a decision procedure to eliminate extra copies.

The reference count, that keeps track of the number of references at runtime, requires extra work to ensure atomic update on multi-threading environment. Unlike reference counting, our approach uses a boolean flag instead, to indicate who is responsible for deallocation.

Memory Deallocation. `Rust` programming language[4] uses *single owner rule* and *move semantics* to avoid dangling pointers. Our deallocation analysis has similar single invariant but uses tree-based structure flag. The flag is pointed to a single top-level variable of a structure or array. Once the variable is deallocated, all its allocated memory and substructures are dropped as well.

`C++11` also provides smart pointers[3] to reduce the misuse of pointers by automatically garbage collecting at runtime. In particular, the single ownership of *unique pointer* is similar to our single deallocation. But the raw pointers used in our project are different in a number of ways.

Unlike the strict copy prevention in the unique pointer, our raw pointers can be copied or aliased to other pointers as

long as the invariant is maintained by using our deallocation flags and macro system.

The unique pointer becomes empty once it is transferred out to another pointer. But our raw pointers can still access the shared memory space after their flags are taken out and transferred to another variable. This feature is useful when a parameter is passed to a function call and its value can not be deallocated at calling function. Our raw pointers with false flags act more like *weak pointers* as they do not have effects on the destruction of shared memory.

5. CONCLUSIONS

Our naive C code generator already translated Whiley programs into efficient C programs that were orders of magnitude more efficient than the Whiley code generator that generates naive Java code[10]. But it had memory leaks and scalability problems.

This paper shows that static analysis can be used to further improve the efficiency of the generated C programs by eliminating most unnecessary copying and memory leaks. More importantly, our combined static analysis and deallocation flags allow each memory block to be deallocated exactly once, which eliminates all double freeing problems and avoids most memory leaks. This is important for long-running programs on small embedded computers with limited memory.

6. ACKNOWLEDGMENTS

Thanks to Dr David J. Pearce for support on the Whiley compiler and benchmark programs.

7. REFERENCES

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles, Techniques*, chapter 9, page 608. Addison wesley, 2006.

[2] J. Aldrich, V. Kostadinov, and C. Chambers. Alias Annotations for Program Understanding. In *ACM SIGPLAN Notices*, volume 37, pages 311–330. ACM, 2002.

[3] A. Alexandrescu. *Modern C++ design: generic programming and design patterns applied*. Addison-Wesley, 2001.

[4] J. Blandy. *Why Rust? Trustworthy, Concurrent Systems Programming*. Number 978-1-491-92730-4. O'Reilly Media, Inc., First edition, September 2015.

[5] N. Lameed and L. Hendren. Staged static techniques to efficiently implement array copy semantics in a MATLAB JIT compiler. In *Compiler Construction*, pages 22–41. Springer, 2011.

[6] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, volume 42, pages 89–100. ACM, 2007.

[7] D. J. Pearce. Integer range analysis for Whiley on embedded systems. In *Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW), 2015 IEEE International Symposium on*, pages 26–33. IEEE, 2015.

[8] D. J. Pearce and L. Groves. Designing a verifying compiler: Lessons learned from developing Whiley. *Science of Computer Programming*, 113:191–220, 2015.

[9] P. Schnorf, M. Ganapathi, and J. L. Hennessy. Compile-time Copy Elimination. *Software: Practice and Experience*, 23(11):1175–1200, 1993.

[10] M.-H. Weng, M. Utting, and B. Pfahringer. Bound Analysis for Whiley Programs. *Electronic Notes in Theoretical Computer Science*, 320:53–67, 2016.

APPENDIX

A. BENCHMARK WHILEY PROGRAMS

The source code of un-modified Whiley programs can be found at Whiley benchmarks repository (https://github.com/Whiley/WyBench).

```
// Reverse an integer array
function reverse(int[] ls) -> int[]:
    int i = |ls|
    int[] r = [0; |ls|]
    while i > 0 where i <= |ls| && |r| == |ls|:
        int item = ls[|ls|-i]
        i = i - 1
        r[i] = item
    return r
```

Listing 1: Reverse

```
// A square on the board is blank, a circle or cross.
type Square is (int x) where x == BLANK ||
                             x == CIRCLE ||
                             x == CROSS
type Board is (null |{
    nat move,
    Square[] pieces // 3 x 3
} this)// A board has 9 squares, and a move counter
method main(System.Console sys):
      int repeat = 0
      while repeat < max:
          Board b1 = EmptyBoard()
          Board b2 = EmptyBoard()
          int i = 0
          while i < |GAME|:
              int p = GAME[i]
              if p <0 || p > 9:
                  break
              else:
                  if b1 != null:
                      b1.pieces[p]=CIRCLE
                      b1.move = b1.move + 1
                      b2 = b1
                      b1 = null
                  else:
                      if b2 != null:
                          b2.pieces[p]=CROSS
                          b2.move = b2.move + 1
                          // Move to next player
                          b1 = b2
                          b2 = null
              i = i + 1
      repeat = repeat + 1
```

Listing 2: TicTacToe

```
function bubbleSort(int[] items) -> int[]:
 // ...
 int last_swapped = 0
 // Until no items is swapped
 while length > 0:
  last_swapped = 0
  int index = 1
  while index < length:
  //Check previous item > current item
   if items[index-1] > items[index]:
    // Swap them
    int tmp = items[index-1]
    items[index-1] = items[index]
    items[index] = tmp
    last_swapped = index
   //End if
   index = index + 1
  // Skip remaining items, since they are ordered
  length = last_swapped
 return items
```

Listing 3: Bubble Sort

```
function mergesort(int[] items, int start, int end)
                -> int[]:
    if (start+1) < end: // Check the length is > 1
        // Recursively split left array ...
        int[] lhs = Array.slice(items,start,pivot)
        lhs = mergesort(lhs, 0, pivot)
        // Recursively Split right array ...
        int[] rhs = Array.slice(items,pivot,end)
        rhs = mergesort(rhs, 0, (end-pivot))
        // Merge left and right lists into items
        while i < (end-start) && l < (pivot-start) &&
                    r < (end-pivot):
            if lhs[l] <= rhs[r]:
                items[i] = lhs[l]
                l=l+1
            else:
                items[i] = rhs[r]
                r=r+1
            i=i+1
        // Tidy up left and right subarray ...
        while l < (pivot-start):
            items[i] = lhs[l]
            i=i+1
            l=l+1
        // Tidy up right subarray
        while r < (end-pivot):
            items[i] = rhs[r]
            i=i+1
            r=r+1
    return items
```

Listing 4: Merge Sort

```
// This matrix is represent with 1D array
type Matrix is ({
    int width,
    int height,
    int[] data // data[i*width+j] := data[i][j]
} this)

function mat_mult(Matrix a, Matrix b) -> (Matrix c)
requires a.width == b.height
ensures c.width == b.width && c.height == a.height:
    nat width = b.width
    nat height = a.height
    int[] data = [0;width*height]
    int[] a_data = a.data
    int[] b_data = b.data
    int i = 0
    while i < height:
       int j = 0
       while j < width:
          int k = 0
          while k < width:
             // c[i][j] += a[i][k] * b[k][j]
             data[i*width+j] = data[i*width+j] +
               a_data[i*width+k] * b_data[k*width+j]
             k = k + 1
          j = j + 1
       i = i + 1
    return matrix(width, height, data)
```

Listing 5: Matrix Multiplication