Efficient Compilation of a Verification-friendly Programming Language

A thesis

submitted in fulfilment

of the requirements for the Degree

of

Doctor of Philosophy in Computer Science

at

The University of Waikato

by

Min-Hsien Weng

THE UNIVERSITY OF
WAIKATO
Te Whare Wānanga o Waikato

2019

Abstract

This thesis develops a compiler to convert a program written in the verification-friendly programming language Whiley into an efficient implementation in C. Our compiler uses a mixture of static analysis, run-time monitoring and a code generator to find faster integer types, eliminate unnecessary array copies and de-allocate unused memory without garbage collection, so that Whiley programs can be translated into C code to run fast and for long periods on general operating systems as well as limited-resource embedded devices. We also present manual and automatic proofs to verify memory safety of our implementations, and benchmark on a variety of test cases for practical use. Our benchmark results show that, in our test suite, our compiler effectively reduces the time complexity to the lowest possible level and stops all memory leaks without causing double-freeing problems. The performance of implementations can be further improved by choosing proper integer types within the ranges and exploiting parallelism in the programs.

Acknowledgements

This work can not be done without the team of my wonderful supervisors. It is fantastic to have the opportunity to work with them in such a good department.

I would like to thank my supervisors: Dr. Robi Malik, Dr. Mark Utting and Dr. Bernhard Pfahringer for contributing to most of the thoughts and insightful feedback in this project. They consistently give me lots of support on thesis writing and under their guidance, we explore two difficult research fields — compiler optimisation and program verification — and overcome lots of problems and have several publications about our findings.

My sincere thanks also goes to Oracle Labs, Australia for providing travel funding to attend SAPLING14 and SAPLING16 meetings.

With a special gratitude to Dr David J. Pearce and the people involved in the development of Whiley compiler at the School of Engineering and Computer Science, Victoria University of Wellington.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software in modern life is used anywhere and anytime, so bugs occur consequently. A single software failure can lead to severe and costly losses as it requires a long correction time, extra efforts for debugging and software patch, and most importantly can cause damage to people's productive life.

The software bug problem becomes worse as the increasing software complexity rapidly raises the difficulty of debugging, and also the bugs from poor coding expose potential risks to security vulnerability and cause crashes in systems.

Tools for improving software quality are needed and can be developed in different ways. *Testing-based methods* use a set of test cases to check if software meets its requirements and find possible software defects. *Software verification* (Huth and Ryan, 2004) applies formal proof techniques to show a program works correctly and verify the software is fit for use. The formal and rigorous proofs also help programmers come up with precise and reliable program design, and guarantees the correct performance within proper use so that potential life-threatening errors in safety-critical systems can be eliminated.

However, it is a grand challenge to build a verifying compiler (Hoare, 2003) using automated mathematical and logical reasoning to find as many bugs as possible at early compilation. Much research has attempted to enable the verification and reduce design flaws for existing programming languages,

e.g. Extended Static Check for Java programs (Flanagan et al., 2002) and Spec# (Mike Barnett, 2005) for C# programs.

Whiley (Pearce and Groves, 2015a) is a new programming language and is designed with an extended verifying compiler to make it easy to write up formal specifications in the program and verify the software at compile-time such that the program can run correctly without run-time errors. The Whiley compiler can also convert the program into Java or JavaScript to be executed across heterogeneous platforms.

This thesis focuses on building a compiler to translate high-level Whiley into low-level C code and improve the efficiency of generated C code, and formalises and proves the memory safety of the generated code using formal verification. However, as value semantics is used in Whiley to guarantee program correctness, the naive and line-by-line translated implementation has potential inefficiency problems as follows.

- Arbitrary-sized integers leads to poor performance.
- Too much extra and unneeded array copying increases memory overhead and lowers the efficiency.
- Manual deallocation is required to avoid memory leaks and to ensure memory safety.

Since Whiley is intended to be used for programming embedded devices as well as general programming, an inefficient implementation like this is not acceptable, as it would mean Whiley programs running on small embedded processors could run out of memory, or the program might run too slowly for its intended purpose. For general acceptance of Whiley, it is important that reasonably efficient implementations of Whiley are available.

This leads to the main research questions of this thesis:

> Can the overheads caused by the design of Whiley be reduced using a compiler whilst preserving the correctness?

Contributions The main contribution of this thesis is as follows.

- Building the optimising compiler for Whiley and preserving the safety.
- Inventing algorithms for copy eliminations and improving the efficiency.
- Inventing new algorithm for de-allocation analysis:
  - Combines static and dynamic analysis,
  - Guarantees exactly one memory de-allocation (no leaks nor double freeing problems occur in the generated C code),
  - Has less overhead than reference counting.
- Proving the memory safety of our macros using formal verification.

The below describes our compiler back-end in more detail.

- Abstract interpretation-based bound inference with extended symbolic analysis is developed to estimate the intervals of integer variables and speed up the convergence of approximating the ranges within finite steps. It also finds the matching patterns and make any necessary program transformation for high efficiency of the resulting code. A shorter version of bound analysis also appears in the paper (Weng et al., 2016).
- The value semantics in Whiley makes copies at each assignment and function call, so wastes time and memory copying large arrays. A copy elimination analyser is developed to detect and remove unnecessary copies wherever possible. By reducing expensive overheads of array copying, the generated code gains speed-ups and has more memory space to run on large-scaled problems. A shorter version of copy elimination analysis is presented as the conference paper (Weng et al., 2017).
- Memory deallocation without garbage collection is complex particularly for aliased and shared memory. A deallocation analyser extended with dynamic run-time monitoring is developed in this thesis. It inter-operates

with the copy analyser to find unneeded memory when no longer used, and inserts deallocation macros in the generated code to avoid memory leakages and ensure each memory space is freed only once. A shorter version of memory deallocation analysis also appears in the same conference paper as copy elimination analysis (Weng et al., 2017).

- Semi-formal proofs are constructed to show our deallocation eliminates double deallocation problems and avoids memory leaks in the generated code whilst preserving memory safety. The proofs involve assumptions, invariants and the program reasoning about pre-and post-conditions, and are converted into Boogie program and are validated using the automatic SMT theorem prover Z3 (version 4.6.1).

- A code generator is developed to automatically translate the source Whiley program at byte-code level into the C programming language. The code generator can use our copy or deallocation analysis separately, or combine the results of both copy and deallocation analyses to improve the generated code, such that the resulting C code runs efficiently.

- Our code analysis and code generator have been applied to five micro-benchmark programs and four large case studies. The results show our code optimisation can remove most of unnecessary array copies in complex programs and give high efficiency. Furthermore, our optimised code can absolutely prevent memory leaks and avoid use-after-free memory vulnerability without garbage collection.

- Parallel computing utilises multiple processors in modern computers to run the program simultaneously and reduce long waiting time caused by the sequential execution. We convert our generated C code into parallel code by-hand and experiment with three kinds of parallel techniques: Polly compiler (Polyhedral optimisations for LLVM), Cilk Plus task parallelism and OpenMP map-reduce program styles. We provide several case studies of the effectiveness of each technique.

Project Scope Our project aims to reduce the overheads caused by the language design. Our compiler takes a Whiley program as input, analyses and optimises the program at Whiley intermediate representation level (WyIL) to produce a safe and efficient C implementation for running correctly, faster and for longer.

Our project implements a subset of the Whiley programming language with static code analysis tools along with an automatic code generator, and our project limitations are as follows.

- The program can be run with one dimensional array of primitive types (integer, byte and Boolean) without cyclic references. For multi-dimensional arrays, recursive data type or nested data structures, whose sizes and memory space dynamically change at run-time, a re-design of memory deallocation is required.

- The program does not have recursion because our analyser has not yet defined static analysis behaviours of recursive function calls, which can be implemented by extending our analyser as a part of the future work.

- The program invariant and verification conditions are all stripped off when translating into C code, as they are not relevant to the computation part of the program and this kind of code erasing technique is also used in `F*` to `C` code (Protzenko et al., 2017a). These formal specifications can provide crucial and useful information to our compiler, e.g. loop invariant contains the estimated array sizes and the bounded values, and can be encoded as constraints to improve the precision of our static analysis and make a better decision for code translation.

- The program executes in sequential, and does not utilise any concurrency. We are aware that the support of parallelisation is an important future work to gain further performance and better throughput from multi-core machines as well as high performance computing cloud.

Thesis Outline Chapter 2 gives background knowledge. Chapter 3 contains related work that our project uses. Chapter 4 describes our bound analyser. Chapter 5 details our copy elimination analyser, and Chapter 6 describes the memory deallocation approach and provides semi-formal proof of memory safety in our macro design. Chapter 7 presents the procedure of code generation and optimisation. Chapter 8 shows performance evaluation of our code optimisation with micro benchmarks as well four real case studies. Chapter 9 investigates parallelism and gives experimental results on our parallel C code. Chapter 10 gives conclusion and future work.

Chapter 2

Background Knowledge

This chapter provides basic preliminaries for our project.

2.1 Verifying Compiler

Prof. Sir Tony Hoare (the ACM Turing Award Winner, FRS) (Hoare, 2003) once said that it is a grand challenge for computing research to create a verifying compiler, with automated mathematical and logical reasoning, to detect the software errors at the compile time. By catching more bugs at compile-time, we can avoid unexpected software failure while running the program. Also, via the verification process we can check whether the implementation meets user specification, and thus improve the quality of software.

Many researchers have been trying to build up automatic compile-time verifying tools to transform a program into constraints, and verify their validity to prove the correctness of program and identify defects. However, these new tools are extended from object-oriented programming languages (Java and C#) to include verification feature and there are limitations on the usage of a verifying compiler.

2.2 Whiley Language

Whiley (Pearce and Groves, 2015b) is a new verification-friendly functional programming language and its compiler aims to solve the verification issues that arise from object-oriented programming languages. The language uses hybrid functional core and imperative paradigms. The functional core ensures the output of each Whiley function depends only on input values and does not cause any side effect, e.g. $sin(x)$ function always produces the same output value for the same x each time. The imperative layer allows Whiley programmers to describe a program with sequence of statements. Whiley supports:

Pure Function Java or C# language allows functions to have different states, e.g. passing call-by-reference parameter to called function. Because callee may change the value of passed parameter, it would produce different results at each function call.

Side effects are not easily observed by verifying compiler because side-effecting function would modify a variable outside its scope and cause an unexpected error. Whiley (Pearce and Groves, 2015b) explicitly defines functions that are side effect-free and pure, whilst method are impure. Consider the below example.

```
1 function func(int[] a) -> int[]: // Pure function
2   a[0] = 10
3   return a
4 // Impure method
5 method main(System.Console sys):
6   int[] a = [0, 0, 0] // a[0] = 0
7   int[] b = func(a) // Does not update array 'a'
8   assert a[0] == 0
9   assert b[0] == 10
```

Function *func* uses call-by-value semantics and thus does not change the value of input array a, because a is first copied and then passed to called function. The output array b however has updated value. A pure Whiley function has below properties:

- Given the same input, Whiley function always produces the same output.
- Function evaluation in Whiley does not cause any side effect.

Separating pure functions from methods allows specifying what can be undertaken in a function, and simplifies the reasoning and verification of Whiley programs.

Value Semantics Java arrays or objects are passed by reference to the called function, and because both callee and caller can change its value, these objects are no longer immutable. The presence of mutable collections makes it difficult to verify the program as anticipated.

Whiley (Pearce and Groves, 2015b) uses value semantics on compound data types, e.g. arrays, so the verification in Whiley can focus on the values, rather than objects themselves. For example, an array assignment in Whiley copies the value of an existing array, and then assigns to the new variable, so that any change to new array will not affect or update the existing array.

Consider the following Whiley program:

```
1 function func(int[] a) -> int[]:   // a[0] = 0
2     int[] b = a          // b = COPY(a)
3     b[0] = 1             // b[0] = 1
4     assert a[0] == 0
5     assert b[0] == 1
6     return a // The value of 'a' remains unchanged.
```

Variable a and b are both integer arrays. The assignment in line 2 copies the value of array a to b, so variable b does not share the same array as a but points to a new and separate array. Any change to array b will not update array a or return value. As such, value semantics makes function *func* pure because it passes parameters by value and does not cause any change to the actual parameters outside function scope.

Value semantics and pure functions enable Whiley language to have hybrid characteristics of imperative and functional languages. That means, we can write Whiley programs in imperative statements and still ensure program safety using side effect free function.

Unbound Arithmetic The unbounded integers in Whiley (Pearce and Groves, 2015b) can ease the difficulty of reasoning about soundness of arithmetic op-

erations using an automatic theorem prover. For example, adding two 32-bit integers may exceed the maximal value which a 32-bit integer can hold, and thus such an arithmetic will have integer overflow problems and lead to an unpredictable system behaviour.

Whiley verifying compiler can detect bugs at compile-time and convert the program into bug-less Java or C code. However, translating high-level Whiley programs into efficient implementations has some challenges, for example, array copies and unbounded integers causes substantial slowdown on the performance of Whiley implementations.

2.3 Whiley Intermediate Language

Our code analysis first uses Whiley compiler to compile a source Whiley program into WyIL (Whiley Intermediate Language) code and then performs code analysis on WyIL code and translates WyIL to optimised C code.

WyIL byte-code language (Pearce, 2015b) is a register-based and three-address like code, similar to LLVM (Low Level Virtual Machine), with semi-structure control-flows. The three-address form consists of an instruction and three registers. Each register is denoted with a prefix % and an integer number, and the set of register numbers is unlimited to accommodate all operands. A WyIL code has below features:

- A WyIL code statically assigns a register to hold a parameter on entry, constant, local variable or a temporary operand which is used to store computed results. Register number starts from input parameters to all operands in the context order of WyIL code, e.g. register *%0* represents the first parameter, and *%1* maps to the second parameter, etc. And different registers never share the same number.

- Register allocation at WyIL level generates a temporary operand to store the value of computed result. For example, `add %6 = %2, %5` adds the values of register *%2* and *%5*, and then assigns the result to target

register $\%6$, which differs from any other existing ones. By having a unique target register, we can avoid potential variable aliasing at an assignment and a function call.

- Each WyIL code has at most one register on the left-hand side but may contain two or more registers on the right side. For example, a loop bytecode `loop (%3, %4, ...)` lists what registers can be changed within the loop.

WyIL acts as an intermediate language and aims to be translated and optimised into different kinds of implementations and run efficiently across platforms. The WyIL code keeps all type information and preserves all invariant at source code to ensure program behaviour, e.g. we can place a loop invariant to ensure our loop counter does not exceed the maximal loop bound and avoids potential out-of-range error. Also, the WyIL code reduces the number of code types to represent statements and expression in Whiley source code, so that the complexity of our code generation and optimisation can be reduced.

There are a number of WyIL code types. We will choose some code types necessary to our project and illustrate each code type with an example.

2.3.1 Example

```
1  // input: input array, output: output array
2  function func(int[] input) -> (int[] output):
3    int n = |input| // Get the size of 'input' array
4    output = [0;n]// Create output array of size 'n' filled with 0
5    int i = 0
6    while i < n where i <= n:
7      output[i] = input[i] * 2// Array update
8      i = i + 1
9    return output
10
11 // Main entry point
12 method main(System.Console sys):
13   int[] a = [1;20]// Create an input array of size 20 filled with 1
14   int[] b = func(a)// Call 'func' function
15   assert a[0] == 1 // Check 'a[0]'
16   assert b[0] == 2 // Check 'b[0]'
17   sys.out.println(b[0])// Print out 'b[0]'
```

Listing 2.1: Example Whiley program

Example 2.1 *Function func takes an array as input, and creates output array with the length of passed input array, and populates the output array by using a while-loop. Main method creates the input array and makes a call to function func. Then it checks the input and output arrays with two assertions, and prints out the array value.*

```
1  private function func(int[]) -> (int[]): // %0: input, %1: output
2  body: // Function body
3    lengthof %4 = %0 : int[] // %4 = |input|
4    assign %2 = %4  : int    // %2 = n = %4
5    const %5 = 0 : int
6    arraygen %6 = [%5; %2] : int[]  // %6 = [0;n]
7    assign %1 = %6  : int[]         // %1 = output = %6
8    const %7 = 0 : int
9    assign %3 = %7  : int           // i = 0
10   loop (%1, %3, %8, %9, %10, %11, %12) // Start of loop
11     invariant // Start of loop invariant
12       ifle %3, %2 goto label0 : int // 'i<=n'
13       fail
14     .label0
15       return// End of loop invariant
16     ifge %3, %2 goto label1 : int // loop condition 'i>=n'
17     indexof %8 = %0, %3 : int[] // %8 = input[i]
18     const %9 = 2 : int
19     mul %10 = %8, %9 : int // %10 = input[i] * 2
20     update %1[%3] = %10 : int[] -> int[] // output[i] = %10
21     const %11 = 1 : int
22     add %12 = %3, %11 : int  // %12 = i + 1
23     assign %3 = %12  : int   // i = %12
24     // End of loop
25 .label1 // Loop exit
26   return %1 // return output
```

Listing 2.2: Function *func* at WyIL Level

Function *func* Consist of function declaration, function body and pre- and post-conditions. Each WyIL code contains the code itself and includes type information of all relevant operands and results. Because outputting all contents is quite lengthy and hard to interpret, Listing 2.2 displays each WyIL code in a simplified format with selected type information.

Our code generation skips the translation of pre and post conditions because these have been verified during the compilation at Whiley source level, and focus on function declaration and body.

Function Declaration Include function signature and variable declaration. The signature consists of function name, return type and a list of parameter types. In our example, `private function func(int[]) -> (int[])` means the input and output of function *func* are integer arrays.

All variables and operations in a function are statically stored with a set of registers, and register order is consistent with the context of WyIL code. In our example, register *%0* denotes the parameter *input*, and *%1* represents array *output*, which both appear in the function signature.

A register could be associated with a present variable at Whiley source code if it stores the value, e.g. register *%2* maps to variable n.

Function Body Contains a block of WyIL code to represent each statement in Whiley program. The code types used in Listing 2.2 are discussed as follows. The *lengthof* code loads array parameter *input* from register *%0* and writes its array size to temporary *%4*. The *assign* code copies array size to target *%2* or local variable n. The *const* code loads constant value 0 to register *%5*. And the *arraygen* code loads size from *%2* and the value at register *%5*, and then creates an array of the given size and fills each array item with the value, and assigns to a temporary register *%6*. Then by using *assign* code, we can copy array at register *%6* to *%1* or return variable *output*. Similarly, we use *const* and *assign* code to write 0 to register *%3* or variable i.

```
1  loop (%1, %3, %8, %9, %10, %11, %12)// A list of modified registers
2    invariant// Start of loop invariant 'where i <= n'
3      ifle %3, %2 goto label0 : int // i <= n
4      fail
5    .label0
6      return// End of loop invariant
7    ifge %3, %2 goto label1 : int // Loop condition 'i>=n'
8    indexof %8 = %0, %3 : int[] // %8 = input[i]
9    const %9 = 2 : int        //%9 = 2
10   mul %10 = %8, %9 : int // %10 = input[i] * 2
11   update %1[%3] = %10 : int[] -> int[] // output[i] = %10
12   const %11 = 1 : int
13   add %12 = %3, %11 : int  // %12 = i + 1
14   assign %3 = %12  : int    // i = %12
15   // End of loop
16 .label1// Loop exit
```

Listing 2.3: Loop WyIL code

The *loop* code (see Listing 2.3) contains a loop block and includes a set of registers to indicate those registers may be changed by the loop body.

The loop *invariant* code in where clause (e.g. `where` $i \leq n$) is represented as a separate invariant block and placed before the loop condition at line 9. The invariant is translated as *conditional* and *fail* code to throw out a run-time error when the condition does not hold. The *conditional* code is prefixed with `if` and a comparing operator to compare the values of two registers and decide whether to go forward to next code or jump to a further *label* code which indicates a position within WyIL code. In our example, `ifge %3, %2 goto label1` checks that $\%3 \geq \%2$. If so, then jump to *label1*. Otherwise, move on to next step. WyIL conditional code is *forward-only branch* because the control flow does not allow call back and backward branches.

After loop condition, we use *indexof* code to access array at a given index and return the value to target register, e.g. `indexof %8 = %0, %3` is equivalent to `%8 = input[i]`. Then we use *binOp* code to perform arithmetic operation on two registers and writes the result to target register, e.g. `mul %10 = %8, %9` is $\%10 = \%8 \times \%9$. We use *update* code to update the array at a specific index with given result, e.g. `update %1[%3] = %10` is $\%1[\%3] = \%10$. And the loop counter i is incremented by one using a combination of *const*, *add* and *assign* code.

Outside the loop, we place *label* code to indicate the loop exit label when the loop iterations stop. And finally, we use *return* code to return the value of target register and stop the function.

Method are impure and different from side effect free functions.

- A method can call another method and allow side-effecting standard input and output stream, such as `print`, but a function can not call a method nor display messages on console.

- Method argument can optionally be passed by reference

- A method may or may not have a return, but a function always returns values.

```
1  method main(System.Console sys):// Main entry point
2    int[] a = [1;20]// Create an input array of size 20 filled with 1
3    int[] b = func(a)// Call 'func' function
4    assert a[0] == 1 // Check 'a[0]'
5    assert b[0] == 2 // Check 'b[0]'
6    sys.out.println(b[0])// Print out 'b[0]'
```

Listing 2.4: Main Method in Example Whiley Program

Example 2.2 *Consider our example 2.4 again. In main method we make a call to function func with an input array, and add assertions to check the function input/return and print out an array value.*

```
1  private method main(whiley/lang/System:Console):// %0 = sys
2  body:
3    const %3 = 1 : int
4    const %4 = 20 : int
5    arraygen %5 = [%3; %4] : int[] // %5 = [1;20]
6    assign %1 = %5  : int[] // %1 = a = %5
7    invoke (%6) = (%1) example:func : function(int[])->(int[])// %6 =
       func(a)
8    assign %2 = %6  : int[] // %2 = b = %6
9    assert // Start of 'assert a[0] == 1'
10     const %7 = 0 : int
11     indexof %8 = %1, %7 : int[]
12     const %9 = 1 : int
13     ifeq %8, %9 goto label2 : int
14     fail
15   .label2 // End of assertion
16   assert // Start of 'assert b[0] == 2'
17     const %10 = 0 : int
18     indexof %11 = %2, %10 : int[]
19     const %12 = 2 : int
20     ifeq %11, %12 goto label3 : int
21     fail
22   .label3 // End of assertion
23   fieldload %13 = %0 out : {int[][] args,{method(any)->() print,
       method(int[])->() print_s,method(any)->() println,method(int
       [])->() println_s} out} // %13 = sys.out
24   fieldload %14 = %13 println : {method(any)->() print,method(int
       [])->() print_s,method(any)->() println,method(int[])->()
       println_s} // %14 = sys.out.println
25   const %15 = 0 : int
26   indexof %16 = %1, %15 : int[] // %16 = b[0]
27   indirectinvoke () = %14 (%16) : method(any)->()// sys.out.println
       (%16)
28   return
```

Listing 2.5: Method *main* at WyIL Level

Method Declaration Contain all used registers and their associated types. Because register allocation starts from method arguments, register *%0* in our example is assigned to system console object and ready to display any message.

Method Body Can contain everything in function body (See Listing 2.5). In our example, we have *invoke* code at line 7 to call function *func* with the parameter from register *%1*, and then return the result to target *%6*. Invoke code uses the colon to split the contents of code, and `example:func` indicates the called function and `function(int[])->(int[])` shows the input and return types of called function. Furthermore, invoke code can be used to call the functions in Whiley run-time library, such as `Math.max` or `File.Reader`.

We then use *assert* code to handle an assertion at WyIL level by using *conditional* and *fail* code to ensure a run-time exception is thrown out when the assertion condition is not met (see line 9 to 22 in Listing 2.5).

```
1  public type PrintWriter is { // Nested type inside System.Console
2    method print(any),   // out.print
3    method println(any), // out.println
4    method print_s(ASCII.string), // out.print_s
5    method println_s(ASCII.string) // out.println_s
6  }
7  // System.Console type
8  public type Console is {
9    PrintWriter out, // Output stream method interfaces
10   ASCII.string[] args // command line arguments
11 }
```

Listing 2.6: System.Console Package

After two *assert* code, we use two lines of *fieldload* code to access method `out.println` from register *%0* to target *%14* because the method is nested and associated to *System.Console* object. As shown in Listing 2.6, the console has one field *out* and another field *args*. The *out* field is declared as *PrintWriter* type and contains a list of printing method interfaces whilst the *args* field is an array of ASCII code (numerical presentation of characters).

In our example, the *fieldload* code at line 23 loads *out* field from register %0 to %13 and the contents after colon lists all field types of *System.Console* type, which is surrounded by curly braces, and each field is split by comma.

Similarly, the *fieldload* code at line 24 loads *println* field from %13 to %14 and displays field types after the colon.

In line 27, we use *indirectinvoke* code to indirectly call 'println' method as the called method/function is determined by a register. In our example `indirectinvoke () = %14 (%16)` loads `sys.out.println` method from register *%14* and invokes the method to print out passed parameter *%16*. The `method` after colon indicates the types of called method (`sys.out.println`).

A function call in WyIL code can be direct or indirect. The *invoke* code directly runs a static function or method declared in the same source file or method in Whiley runtime library (e.g. `Math.abs`) whereas the *indirectinvoke* code executes a function or method indirectly determined by a given operand.

2.3.2 WyIL Code Types

We categorise and list the WyIL code types with the support level of our project: full, partial and none. The symbols in the table are described as follows. $\mathbf{l_1}$ is target register on the left-hand side. $\mathbf{r_1}$ and $\mathbf{r_2}$ are the operand registers on the right-handed side. **constant** number is the constant value. And **label** identifier indicates a labelled position at WyIL code, **type** denotes a given type and **field** presents a field name of a structure. And **func** is the name of called function.

Table 2.1: Partially supported WyIL code types

Code Type	Description	Syntax
`Assert`	Assertion block	**assert**
`Dereference`	Dereference a reference	**deref** $\mathbf{l_1} = \mathbf{r_1}$
`FieldLoad`	Load a field value from a key	**fieldload** $\mathbf{l_1} = \mathbf{r_1}$ **field**
`IfIs`	Type checking on a register	**ifis** $\mathbf{r_1}$, **type goto label**
`Invariant`	Loop invariant	**invariant**
`NewRecord`	Create a object structure	**newrecord** $\mathbf{l_1} = (\mathbf{r_1}...)$

Table 2.2: Non-supported WyIL code types

Code Type	**Description**
`Convert`	Convert a value to a type
`Debug`	Print out debugging messages
`Invert`	Bit-wise Inversion
`Lambda`	Lambda expression
`Move`	Move a register to another and make the original register void. This *move* is similar to move semantics in Rust language.
`NewObject`	Create an object
`Not`	Invert a boolean
`Quantify`	Encoded quantifiers at WyIL
`Switch`	Multi-way branches
`Void`	Make a register void

Table 2.1 shows a list of partially supported WyIL code types. Our project does not translate *assert* and *invariant* code into C code as a default action, but provides `ea` compiler option to enable its code generation. For structure related code (*dereference*, *fieldload* and *newrecord*), our project supports the code generation of single-array like structure, which contains only one integer array with a few extra integer fields, but our deallocation analysis does not guarantee the memory leaks and safety of structure types. Our *ifis* code checks if a register is null type and can not perform the check on other types. Table 2.2 lists the code types that our project has not supported yet, and the below table shows the code types of fully supported Whiley intermediate level (WyIL) and gives a short explanation of code syntax.

Table 2.3: Fully supported WyIL code types

Code Type	Description	Syntax
`ArrayGenerator`	Generate an array	**arraygen** $\mathbf{l_1} = [\mathbf{r_1}; \mathbf{r_2}]$
`Assign`	Assignment	**assign** $\mathbf{l_1} = \mathbf{r_1}$
`BinOp`	Arithmetic operations	**add** $\mathbf{l_1} = \mathbf{r_1},\ \mathbf{r_2}$ **sub** $\mathbf{l_1} = \mathbf{r_1},\ \mathbf{r_2}$ **mul** $\mathbf{l_1} = \mathbf{r_1},\ \mathbf{r_2}$ **div** $\mathbf{l_1} = \mathbf{r_1},\ \mathbf{r_2}$ **rem** $\mathbf{l_1} = \mathbf{r_1},\ \mathbf{r_2}$
`Const`	Load a constant	**const** $\mathbf{l_1}$ = **constant**
`Fail`	Throw an exception	**fail**
`Goto`	Jump to a label position	**goto label**
`If`	Conditional branch	**ifeq** $\mathbf{r_1},\ \mathbf{r_2}$ **goto label** **ifneq** $\mathbf{r_1},\ \mathbf{r_2}$ **goto label** **iflt** $\mathbf{r_1},\ \mathbf{r_2}$ **goto label** **iflteq** $\mathbf{r_1},\ \mathbf{r_2}$ **goto label** **ifgt** $\mathbf{r_1},\ \mathbf{r_2}$ **goto label** **ifgteq** $\mathbf{r_1},\ \mathbf{r_2}$ **goto label**
`IndexOf`	Access array item	**indexof** $\mathbf{l_1} = \mathbf{r_1},\ \mathbf{r_2}$
`IndirectInvoke`	Indirect function call	**indirectinvoke** $(\mathbf{l_1}) = \mathbf{r_1}(\mathbf{r_2}...)$
`Invoke`	Function Call	**invoke** $(\mathbf{l_1}) = (\mathbf{r_1}...)$: **func**
`Label`	Label position	.**label**
`LengthOf`	Array size	**lengthof** $\mathbf{l_1} = \mathbf{r_1}$
`Loop`	Loop block	**loop** $(\mathbf{r_1}...)$

Continued on next page

Table 2.3 Fully Supported WyIL Code Types (Continued)

Code Type	Description	Syntax
`NewArray`	Create an array from a list of initial values	**newlist** $\mathbf{l_1} = (\mathbf{r_1}...)$
`Nop`	Non-operation	**nop**
`Return`	Return from a function	**return** $\mathbf{r_1}$
`Update`	Update an array	**update** $\mathbf{l_1}[\mathbf{r_1}] = \mathbf{r_2}$
`UnaryOperator`	Unary operation	**neg** $\mathbf{l_1} = \mathbf{r_1}$

2.3.3 Benefits of WyIL Code

Our code analysers and optimiser operate at Whiley intermediate language level because WyIL code provides several advantages over source code. Firstly, WyIL reduces the number of operation code (opcode) types and replaces nested control-flows with uniform branching, so that we can use the same approach as conditional to handle with nested control-flow `break` or `continue`. As a result, the implementation complexity of code analysis can be reduced. Secondly, WyIL breaks down a long calculation into a sequence of binary operations and provides greater flexibility for our back-end to apply code optimisation. Finally, we can take the same WyIL code without needing re-compilation from source code to experiment with different code optimisations and to compare the performance improvement.

2.4 WyIL To C

Our project translates WyIL code into efficient C code of lower memory usage and faster execution, compared to the naive C code that our compiler produces without any optimisation. Two additional things are required to undertake during code generation, as follows.

2.4.1 Bounded Integer

Arbitrary-precision integer requires more memory and computing than a fixed-size integer. For example, *BigInteger* in Java has variable-length size and must run on slow software layer whereas fixed-size integers, such as `int16_t` (signed 16-bit integer), uses exact size and can directly run on fast hardware layer.

In our project we use static bound analysis to find the ranges of integer variables and substitute arbitrary-precision integers with a variety of fixed-size types whenever possible.

2.4.2 Memory Reduction

Unnecessary array copying causes program inefficiency and memory leaks lead to program in-scalability. In our naive implementation of WyIL to C code, we include value semantics to have an array copy at each assignment or function call. But excessive array copies which are not always needed waste execution time and resources. In addition, the amount of memory leaks from heap-allocated arrays is accumulated to cause thrashing and a failure to scale up the program to a larger problem size.

In our project, we design a macro system to detect unnecessary array copies and minimise the memory usage whilst maintaining the memory safety.

2.4.3 System Architecture

Our WyIL-to-C backend includes code generation and three static analysers (integer bound, copy elimination and deallocation analysers). Our backend operates at Whiley intermediate language (WyIL) generated from high-level Whiley source code to generate and optimise C Code.

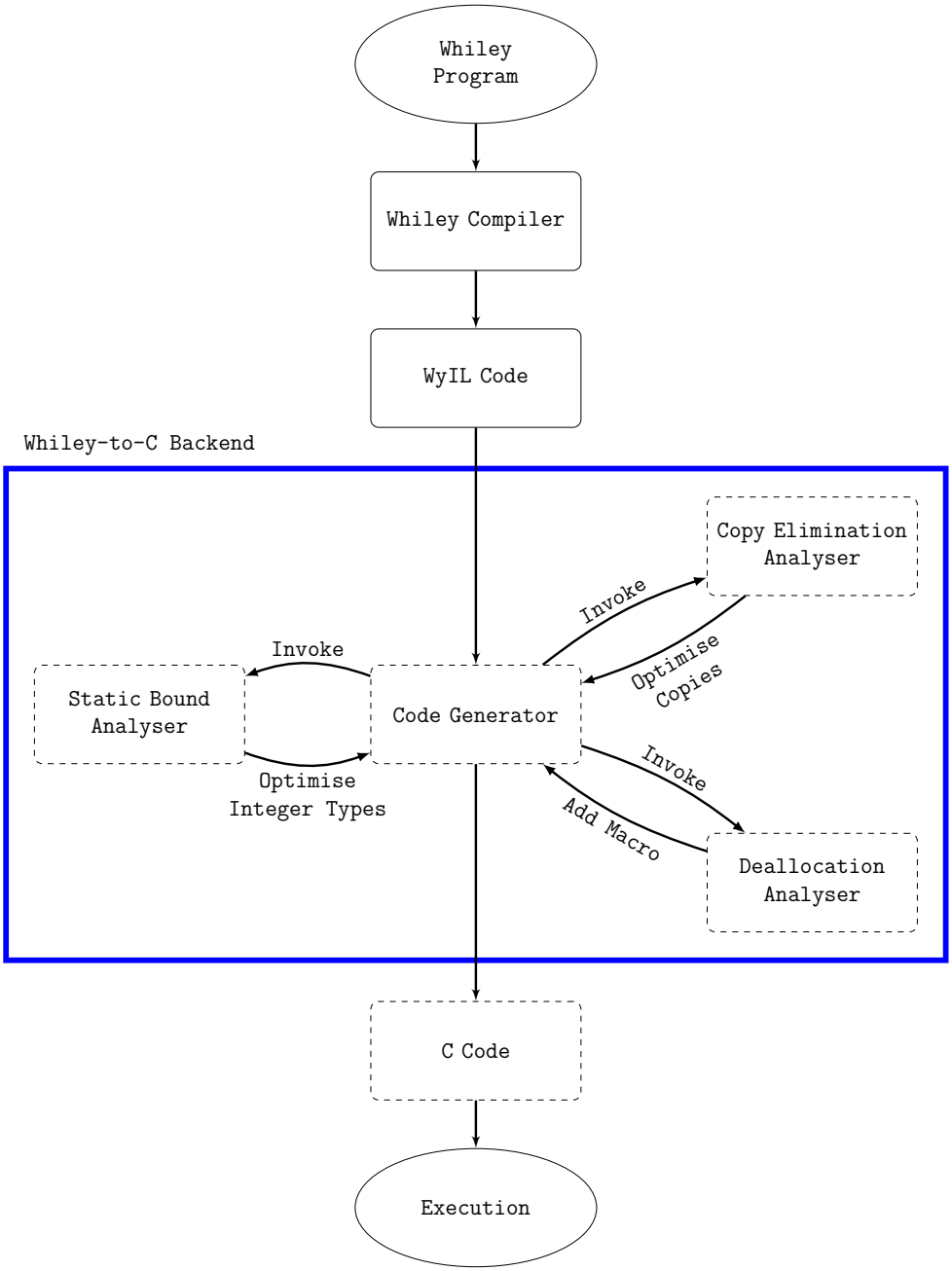


Figure 2.1: System architecture (dashed boxes: our project)

As shown in Figure 2.1, the code generator converts the WyIL code into efficient C code while interacting with bound analyser to make use of the fixed-size integer types, and with copy elimination and deallocation analyser to minimise the memory usage in the generated C code by reducing the number of array copies and de-allocating on the unused arrays. Our project goal is to implement a large subset of Whiley in C with parallelism where possible/useful.

Chapter 3

Related Work

In this chapter we first go through some static (bound) analysis to find a proper tool to estimate integer intervals and choose bounded integer types, and then examine some related work about memory management and design principles to reduce the memory usage. Lastly, we reviewed some important work about static and dynamic analysis to eliminate the unused array copies and improve program efficiency.

3.1 Static Analysis

Static analysis validates the consistency between software specifications and program behaviours using mathematical methodologies. For example, the bound consistency technique is widely used to solve the finite constraint domain problem (Marriott and Stuckey, 1998).

However, the problems of object-oriented program languages, such as side-effects and non-deterministic results, make it a grand challenge (Hoare, 2003) to create a compiler, with automated mathematical and logical reasoning, that can statically verify the specifications and detect the errors at compile-time.

Some automatic static analysers use different approaches to find software defects at early compilation stage to improve program correctness and produce high-quality software. *Extended static checker for Java (ESC/Java)* (Flanagan et al., 2002) uses an automatic theorem prover to analyse the program and find

common Java run-time errors, (e.g. array out-of-bound or null dereference, etc). Also, ESC checker can be used to analyse concurrent Java programs and issue warnings for potential run-time race conditions and dead locks. As ESC requires to annotate specifications in programs, the annotation burden and excessive warning messages could cause inconvenience for programmers.

`Boogie`, which was originally developed in Microsoft Spec# (Mike Barnett, 2005) system to verify a `C#` program, acts as an intermediate verification language (Leino, 2008) to transform a Boogie program into verification conditions. By using an automatic theorem prover (e.g. `Z3` satisfiability modulo theories solver (de Moura and Bjørner, 2008)) it can statically prove the correctness of a program against pre- and post-conditions, and Boogie can point out possible error cause in the program if verification fails. Using Boogie can avoid expensive run-time check and improve the efficiency of program execution as Boogie has statically verified those conditions at compile time and thus can remove them from run-time. Furthermore, Boogie verification resembles writing a program, e.g. we can write *frame conditions* as modifies and ensures clauses in Boogie to restrict which variables a function can change and to write complex formulas in pre- and post-conditions. Apart from Spec#, Boogie supports a variety of programming languages, including Java byte-code with BML (Mallo, 2007), Dafny (Leino, 2010), Eiffel (Tschannen et al., 2011) and C (Vanegue and Lahiri, 2013). Furthermore, Whiley also supports Boogie as a verification back-end (Utting et al., 2017).

The static analysis using abstract interpretation can approximate the abstract semantics of a program without execution and allows the compiler to detect errors and find applicable optimisation. For example, *Microsoft Research Clousot* (Manuel Fahndrich, 2010) can statically check the absence of run-time errors and infer facts to discharge assertions. In our project, the number of WyIL code is much larger than its high-level and human-readable Whiley source code as every complicated operation in Whiley is broken down into a series of three-address forms in WyIL to preserve the semantics. We use

abstract interpretation-based static analysis to analyse such a large amount of WyIL code because it can operate at lower execution time and still produce high precision.

3.2 Static Bound Analysis

Static bound analysis is a compiler optimisation technique, which estimates the upper and lower bounds of a variable and detects potential run-time arithmetic overflows at compile time.

The static bound analysis in *LLVM* (Low Level Virtual Machine) becomes popular as the LLVM code can be optimised and converted to machine-depend assembly code by the compiler without changes to original source program. For example, an industrial-quality range analysis (Campos et al., 2012) has been implemented in LLVM compiler and adapts revised polynomial interval analysis (Gawlitza et al., 2009) to observe the decrease or increase in cycles and then saturate the cycles by using the widening operator. However, LLVM bound analyser tends to have overflow problems as the signedness information has been lost at LLVM level, but could be solved by using a signedness-agnostic bound analyser (Navas et al., 2012).

Static loop bound analysis approximates the number of loop iterations and proves the termination of loop. And the estimated loop bounds can also be used to unroll the loop and to increase program speed. The commonly used techniques include *pattern-matching* and *counter increment*. Pattern-matching *CodeStatistics* (Fulara and Jakubczyk, 2010) can prove the loop termination by finding all for loop patterns in Java programs, and inserting termination conditions as annotation into existing code. Their results show that their method can efficiently prove 80% of for loops and detect error-prone loops in large-scaled applications, including *Google App Engine*, *Apache Hadoop*, *TomCat* and *Oracle Berkeley DB*. A counter-incremented approach (Shkaravska et al., 2010) is presented to obtain the linear and non-linear loop-bound

function (LBF), that binds the numeric loop condition to the number of loop iterations. Shkaravska's approach can handle very complicated loops to infer polynomial LBFs but also ensure the correctness of derived LBFs using an external verifying tool. Due to inefficiency on simple loops, it is usually considered as a complementary approach to other existing ones.

Pattern-matching and counter-increment approaches do not handle multi-path loops of different effects or non-trivial patterns well. A *control-flow refinement* technique (Gulwani et al., 2009) is used to transform a multi-path loop into one or more explicit interleaving loops to simplify the analysis, and then use *progress invariant* technique to compute precise symbolic loop bounds. The experimental results show that their approach can find 90% of loop bounds in a large *Microsoft* product.

The static bound analysis usually has a trade-off between precision and efficiency. When dealing with undecidable problems, the analyser usually accepts imprecise results to avoid long running time and non-termination problems. An interval analysis without widening or narrowing operator (Su and Wagner, 2004) is proposed to solve integer range constraints, and shows that their approach produces precise bounds in polynomial time whilst the termination is guaranteed. Our project uses abstract interpretation iteration strategy to compute the integer bounds in an abstract domain and accelerate the convergence of bound inference by using widening operator with thresholds (Blanchet et al., 2003) which goes through a number of threshold values and effectively approximates the loop bound to fix-points within finite time.

A forward-propagated integer analysis (Pearce, 2015a) is presented in Whiley compiler to exploit type and loop invariant to restrict the ranges of integer variables with explicit integer type declaration. In our project, we use abstract interpretation-based static bound analysis to estimate the ranges of integer variables, and base on the resulting bounds to use precise integer types in the generated code. Our approach targets at abstract typed integers and infers their bounds with abstract interpretation-based widening operator. We may

obtain the over-estimated bound results but ensure there is no integer overflows occurring with our bound results and also guarantee the termination of our bound analysis.

3.3 Memory Management

There are two kinds of memory space: stack and heap, and both stack and heap are stored in the same random-access memory (RAM). Our project represents a Whiley array with heap-allocated array in C and needs to undertake below work to produce efficient C code:

- Extra dynamic memory deallocation is needed to free the arrays on heap.
- Extra analysis for array-typed arguments is required to avoid memory leaks during a function call.
- Extra care must be taken to ensure the aliased array is only freed once and no double freeing memory problem occurs in our program.

We give a brief comparison between stack and heap arrays, and discuss the region-based memory management.

Stack Store small and local arrays faster, because stack memory can be freed automatically without extra deallocation efforts when the function returns.

There are some restriction on stack memory. *i*) The array on stack can be passed to called function, but can not be returned because all stack data will be deleted at function return. *ii*) Stack size is set to be small (8MB) to avoid over-writing heap space. A too large stack requires moving heap space and may invalidate all heap-allocated pointer addresses and break the program. Also, if maximal stack size is reached we have a `stack overflow` and cause segmentation fault. *iii*) Arrays on stack must be declared and specified statically at source code and do not allow re-allocation to grow and shrink back the array size at run-time.

Heap Use dynamic memory allocation to provide more flexibility to store large and variable-length data of longer life-time.

Heap-allocated memory has several advantages over stack one. *i*) Heap arrays can be used outside the function as a parameter or return. *ii*) Heap size is limited to the size of virtual address space (thanks to the operating system's swap mechanism), so is able to accommodate most problem sizes in 64-bit operation system. *iii*) Heap provides several built-in functions for programmers to dynamically change the heap-allocated array size at run-time by using `malloc`, `realloc`, `calloc` and `free` functions.

However, heap space has less efficient allocation than stack and may cause memory leaks and double freeing issues. Region-based memory allocation is another way of memory management.

Region Region-based memory management (Hicks et al., 2004) allocates and assigns each array to a region, and has hybrid advantages of heap and stack memory.

```
1  int* bar(int* a){
2    return a; // Return input array 'a'
3  }
4
5  int* foo(){
6    Region *r1 = createRegion();// Create region memory
7    // Allocate array 'a' to region 'r1'
8    int* a = allocateFromRegion(r1, sizeof(int)*10);
9    int* b = bar(a); // Array 'a' and 'b' are aliased.
10   destroyRegion(r1); // Free aliased array 'a' and 'b', so 'b' becomes null.
11   return b; // Array 'b' is dangling pointer
12 }
```

Listing 3.1: Dangling pointers in region-based memory

Region memory, similar to stack, has low overheads of allocation and deallocation because all the objects in one region are allocated to a block of contiguous memory space, and when the region is destroyed, all objects are deallocated at once in a constant time without needing to empty each object separately. Moreover, region-allocated objects have longer lifetime and larger space access than stack-allocated ones. As such, the region memory is more suitable to store complex data structures, such as linked list, and make it easy

to reason about the required memory space.

However, region memory needs manual de-allocation, like heap, and still has memory leaks and dangle pointers. Consider the example in Listing 3.1. Arrays *a* and *b* are aliased at function call (line 9) but they are freed when region *r1* is destroyed (line 10). As such, function *foo* returns a dangling pointer that refers to invalid address.

Solving this problem requires *region inference* to statically find the scope of variables and limit the use of deallocation, e.g. *unique pointers* are integrated to safe C dialect Cyclone (Hicks et al., 2004) programming language to ensure only one valid reference points to an object, and to avoid any attempt to de-reference any dangling pointer.

Reference counting and garbage collection are common approaches, which are used to deal with the deallocation of unused memory automatically.

3.3.1 Reference Counting

Reference counting algorithm can reclaim an unused memory as soon as it is no longer in use. The basic reference counting firstly creates an extra counter for each referenced item to track the number of its references during execution. Secondly, it increments the counter when a new reference is created and referenced, and decrements the counter when the reference is out-of-scope or over-written. Lastly, the item can be deleted when its counter reaches to zero.

Reference counting gives prompt response to clear out all unused memory and reduce memory usage to improve performance in limited resource systems, particularly embedded system.

However, frequent updates on the reference count consume too much computation and slow down the execution. Also, reference counting can not handle reference cycles, where an object refers to itself and forms a cyclic chain of objects. We could solve this cyclic reference issues by implementing additional approaches to reference counting but increase its complexity. In our project, we focus on only arrays of primitives (no pointers of pointers are allowed) so

there will be no cycles used in the program and thus reference counting can be used to solve our memory deallocation problem. We use a run-time boolean flag, rather than counting reference number, to keep track of reference changes from one variable to another.

3.3.2 Garbage Collection

Garbage collection automatically detects and frees unused memory without manual instruction so garbage collector can avoid some memory leaks and safety bugs, such as dangling pointers and double freeing problem, which frees the memory space that has been de-allocated before.

Tracing garbage collection algorithm identifies in-used and unused objects, which are no longer referenced, and then reclaim unused memory. Unlike reference counting, the garbage collector can effectively free the memory of cyclic reference objects. The basic *mark-and-sweep* algorithm assigns each object with a *flag* to indicate whether the object is *reachable* and build up a set of *roots* to preform two-phased operation to detect all unreachable objects (*mark phase*) and clean the memory space for all unreachable objects (*sweep phase*).

However, mark-and-sweep phase needs to suspend the program during garbage collection and may cause long pause as the algorithm must examine and check the entire memory space. Also, make-and-sweep may consume and exhaust the memory space if it has been triggered constantly. Additional and well-defined methods are required to solve these performance issues. In our project, the target programs do not have cyclic references and therefore there are no needs for automatic garbage collection to clean up memory.

Rust is the most relevant to our project. Rust programming language (Blandy, 2015) provides the control over memory, like C and C++, and also ensures the memory safety and data-race-free concurrency with *single ownership*, *move semantics*, *borrow reference*, etc. So Rust compiler can estimate the lifetime

of every variable and drop every value whenever not having ownership, so that dangling pointers can never be used. Our project bases on Rust design principles to determine the responsible deallocation at run-time and avoid double freeing problem.

Smart pointers (Alexandrescu, 2001) are implemented in `C++` with built-in memory management to reduce the misused pointers and avoid memory leaks. Our project uses similar pointers, particularly *shared pointers*. Multiple pointers are allowed to access the shared memory. But the de-allocation occurs only once as the flag has been transferred to the last (used) variable during assignments.

3.4 Copy Elimination

Copying is an expensive operation and creating redundant copies leads to inefficient problems in most programming languages that uses the copy/value semantics. Some reference type programming languages, including C, C++, Java and Rust (Blandy, 2015), allow programmers to mark the immutable variables as mutable and update the values without copying. However, in a copy-semantics programming language, such as MATLAB, Whiley or TCL (Ousterhout et al., 2010), copies are always made to avoid side effects of updating existing variables, so compiler optimisations have been developed to find unnecessary copies in a program.

Static analysis can be used to detect unneeded copy operation in functional programming languages. Static abstract interpretation reference counting (Hudak and Bloss, 1985) was proposed to approximate the number of references with the termination of inference guaranteed, so that the compiler can apply in-place updates onto the variables which are used only once. However, the copy avoidance on divide and conquer programs, such as *quicksort*, requires a further inter-procedural analysis (Gopinath and Hennessy, 1989). Their approach uses fix-point iterations to compute the aliasing of function argument

and substitute for call-by-reference parameter. Our approach performs a similar inter-procedural and linear-timed analysis to collect the sets of read-write, return and live variables, rather than their exponential-timed aliasing analysis, to make the determination of parameter copies during a call.

Our copy elimination analysis appears most similar to the algorithm of hybrid static analysis and dynamic reference counting (Goyal and Paige, 1998) proposed to eliminate copies in an imperative programming language SETL.

Their approach keeps track of reference counts during program execution but our approach uses boolean run-time flags, which indicate whether the variable is responsible for the deallocation of its memory space, and speeds up the run-time checks. Their approach, like ours, uses static alias analysis and live variables to find destructive updates at each program point and inserts extra code to reduce the reference counts so that the run-time can replace the copy with an in-place update when the reference count reaches one. Our approach relies on live variable analysis to remove the copies of dead variables at compile-time.

Their analysis can run in low polynomial time, but does not perform well on function call parameters. So an efficient and polynomial-time algorithm (Wand and Clinger, 2001) for inter-procedural array update was developed to generate a set of constraints from live variables and aliasing analysis results and solve these constraints to replace call-by-value parameters by the references. Instead of inferring constraints, our approach performs static analysis on both called function and caller sites, and uses a rule-based macro system to explicitly remove or keep the copy of a parameter. But under some uncertain function behaviours, our approach chooses to keep the extra copies of parameters to avoid side effects of function calls and includes dynamic checks to delete unused parameter copies.

MATLAB uses reference counting to determine the unneeded copies but incurs extra run-time overheads and slows down the program execution. Thus, a pure static analysis without reference counting (Lameed and Hendren, 2011)

was developed for the MATLAB JIT compiler. Their approach firstly performs a quick check to remove the copies of read-only variables and secondly uses a forward analysis to find all the required copies for live variables and then performs a backward analysis to find a better location to place the copy. Their approach is pure static analysis but relies on garbage collection to free unused array copies. Our approach combines static and dynamic analysis, and provides a simple way to eliminate unnecessary copies and to undertake the deallocation tasks without garbage collection.

3.5 Verifying Compiler

A verifying compiler (Hoare, 2003) uses automated mathematical and logical reasoning methods to check the correctness of the programs that it compiles. The compiler verifies the program (mostly written in a high-level programming language) by generating all the verification conditions, and discharging each via a built-in or external verifier, such as SMT solver, to find any runtime error when possible, and prove the program correctness. Once the input program is verified, the compiler translates it into the low-level implementation with explicit details (memory model and data representation) to run on the machine.

VCC (Cohen et al., 2009) verifier enables C programs with verification annotations to include functional pre-and post-conditions, and with its static verifier proves the C code at function level. To deal with variable aliasing and dangling pointers in C, VCC introduces its ownership memory model and type invariant, and therefore extra annotation overheads are unavoidable. CompCert (Leroy et al., 2016) compiler also verifies the program correctness in C level using Coq theorem prover. Our approach however verifies high level Whiley programs, rather than the low-level C code, because Whiley is designed to ease the verification difficulties. For example, the use of value semantic makes every value immutable without any aliasing, so the verification in Whiley becomes less complicated than C code.

Many verification frameworks use a similar strategy: verifying the program in high level language and translating into low-level code for better efficiency. Dafny (Leino, 2010) verifier extracts all verification conditions from the source code, and then translates into Boogie (Leino, 2008) and validates the Boogie program using automatic SMT Z3 solver (de Moura and Bjørner, 2008). After the verification, Dafny compiler takes the program and converts into executable C# code. The Dafny compiler (Leino, 2017) uses two strategies to improve the efficiency of generated code. First, it chooses fixed-sized integer types based on given constraints whenever possible, and takes advantage of their fast speed at runtime. Second, it ignores the compilation of specifications (pre-and post-conditions) into actual code, and reduces the overheads. Our approach includes a similar bound analysis to replace the unbounded integers with the smallest fixed-size types. Our analysis erases all the specifications, which are not related to computation, from Whiley programs and also performs extra code optimisations, e.g. array copy elimination and memory deallocation, to reduce the overheads of C code.

F* verification programs (Protzenko et al., 2017b) can be compiled to fast and well-defined C code. Its memory model is similar to CompCert, and can facilitate both the stack and heap with memory safety guarantee. As such, its C code never has out-of-bounds access or double freeing problems, but due to the restriction in F*, the C code requires explicitly manual heap allocation in the source F* program. However, our approach implicitly uses the heap space for all array variables, and can automatically place allocation or deallocation in the generated C code without any statement in Whiley programs.

The optimising compiler has been actively applied on machine learning area of research. Glow compiler (Rotem et al., 2018) at Facebook translates the machine-learning specific programs written in high-level Pytorch Paszke et al. (2017) language down to LLVM code and optimises memory usages and instruction schedules to take advantages of hardware features and execute across various target machines.

3.6 Rust Comparison

Rust compiler (team, 2019) converts its program into LLVM IR code, and by using the `Clang` compiler, the generated LLVM code can be compiled and optimised to fast and safe executables for various target machines. When translating Rust to LLVM code, Rust compiler can use type checker to infer untyped variables and include borrow check to enforce the generated code conforming to the move and borrow semantics in Rust ownership system, so that the LLVM code can be run safely without needing a garbage collection.

Our approach is inspired by Rust ownership but the idea of 'owner' is simplified to deal with memory deallocation only, and we use the below scheme to achieve zero memory leaks and zero double deallocation.

- Every array variable is associated with a Boolean *deallocation flag*. This flag's value is used to keep track of which variable is responsible for actual deallocation of the shared memory space at runtime. Unlike Rust ownership that requires the owner to explicitly gain the read-write access, our flag is only used to decide whether freeing the allocated memory, and has no controls over value mutability.

- Our *deallocation invariant* ensures that at any program point, exactly one variable is responsible to free the allocated memory space. This is similar to Rust single ownership principle.

- Our approach includes 8 *deallocation macros* and ensures our deallocation invariant always holds after each macro. Rust relies on variable scopes to drop out the values, but our approach does not use the scope (every local variable is in function scope) but use static analysis and runtime flag to decide whether to free unneeded memory space.

Chapter 4

Live Variables and Bound Analysis

On 10 January 2017, 22 transactions of *Largan Precision Co.* at Taiwan stock exchange were disruptively halted due to an integer overflow bug on price, which was falsely set up with 32-bit integer range by the system. So, when the maximal value of each Largan stock transaction $(4,295,250,000)$ exceeds the upper limit of unsigned 32-bit integers $(2^{32}-1=4,294,967,295)$, the safety mechanism was accidentally triggered and then caused huge loss to investors. Such a false alarm can be avoided by choosing a proper and suitable integer type, e.g. unsigned 64-bit integers, to increase the stock price range.

This chapter presents an abstract interpretation-based bound inference approach (Weng et al., 2016) to estimate the range for integer variables at Whiley intermediate level and to make use of primitive integer types, rather than third-party infinite integer type (e.g. using GMP arbitrary precision library), on generated code and increase the efficiency.

The Whiley program is first compiled into WyIL code, and then the bound analyser is invoked to estimate the upper and lower bounds of each integer variable and determine a specific fixed-width types (`int16_t`, `int32_t` or `int64_t`) such that the type has the smallest range but still can hold the maximal and minimal value of the variable to avoid arithmetic overflows during execution.

The bound analysis develops a conservative bound consistency technique to ensure that the output bounds are large enough to avoid all integer overflows in the generated code. In addition, the abstract interpretation-based widening operator is used to speed up the converging time of bound inference.

The bound analyser is implemented as a Java plug-in on top of Whiley compiler project. It infers the bounds of integer variables in two phases. First, the analyser evaluates each WyIL code semantics to extracts the constraints on abstract domain. Then the analyser computes bounds with the bound consistency technique and speed up the convergence time by using the abstract interpretation-based widening operator.

4.1 Bound Consistency Check

Bound consistency technique (Marriott and Stuckey, 1998) restricts the variables to a finite set of values and satisfies the arithmetic constraints. This technique allows the bound analyser to propagate lower or upper bounds among variables in the form of constraints and ensure that lower bounds never exceed upper ones.

The bound analyser takes the code of a function block as input, goes through the context-sensitive bound inference procedure, and produces the output bounds of a function call, which reflect the input parameters. Bound inference starts from *main* function and performs inter-procedural bound analysis on a function call whenever necessary. The steps include control flow graph (CFG) construction, live variable analysis and bound inference.

4.1.1 CFG Construction

The analyser builds up *control flow graph* of each function. It scans the code at each program point, processes the semantics of each line of code to construct a new block or get current block and add the code to that block. Each block connects other block with a directed edge to show the program execution flow.

For example, the below while-loop contains three blocks: *loop header*, *loop body* and *loop exit*.

```
1 // While−loop in Whiley
2 while i<10:
3   sum = sum + i
4   i= i + 1
```

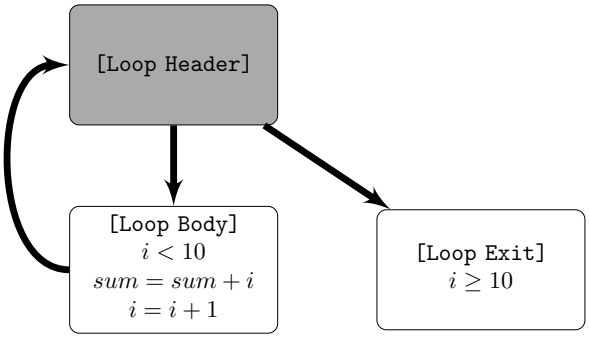


Figure 4.1: While-loop structure

As shown in Figure 4.1, the loop header is an empty block, which does not have any code, but used to connect loop body and exit. The loop body stores the loop condition and other statements at loop body whereas loop exit stores the negated loop condition, and other code after the loop.

Our project supports standard control flow block types (Aho et al., 1986): `basic block`, `entry` and `exit`, loop structure (`loop header`, `loop body` and `loop exit`), `if` branch, `else` branch, `label` and `return` blocks with addition of `update` and `function call`. Entry block is the root node of graph whereas exit is the leaf node, which does not have any child node.

Loop structure and if-else branches are typical blocks as they change the control flow, based on some conditions, and then perform different instructions. A basic block includes a sequence of code which does not branch out the flow. Label code indicates the needs of a new block scope in the current flow, so we create a new block, linking to current block as a child node, to store the code within labelled block after the label code.

Return block represents the end of a program execution path. As a function may contain conditional branches and create more than one execution paths, a function may have multiple returns. All return blocks link to exit block, to indicate the termination of a function.

Apart from above control-flow blocks, we introduce additional update block

and function call block. The *update* code accesses an array item at a given index and updates it with a new value, e.g. $a[0] = 1$. The update code makes changes to an array variable but does not have a copy or aliasing. So a separate update block is needed for live variable analysis to determine the live variables after update code.

A function call with array-typed parameters involves code optimisation, such as array copying and memory deallocation, and thus requires a separate block to check the liveness and function behaviour. Each block consists of:

- *Block name* and *type* along with all the *code* within the block scope.
- *Parent blocks* connecting to the block towards entry, and *child blocks* connecting to the block away from entry.
- *Constraints* that are extracted from each code in the block.
- *Live variable set* which contains the in-use variables after the block, and *dead variable set* which includes the unused variables after the block.
- *Bound set* which contains lower and upper bounds for all live variables in the block.

4.1.2 Live Variable Analysis

Live variables at a control flow block means the variables may be used or read after the code whereas dead variables will not be used in the future. Live variable analysis (Aho et al., 1986), an iterative backward data-flow algorithm, finds and collects live and dead variables before and after each block in a function. Firstly, the analyser constructs control flow graph. Secondly, it backtracks through each block and computes *code-level* liveness transfer equation at each line of *code* at block *blk* in function *func* to find the set of live variables before and after this block, denoted by $IN(blk)$ and $OUT(blk)$ respectively. This procedure repeats until all sets have no changes and fixed-point is reached.

Procedure 4.1 Compute Live and Dead Variables

Input: Function *func* and its control flow blocks
Output: Live and dead variables at each block in function *func*

1: **Variables**
Code c at block *blk* in function *func*
$def(c)$: a set of variables defined at code c
$use(c)$: a set of variables used at code c
$in(c)$: a set of live variables before code c
$out(c)$: a set of live variables after code c
$VARS(blk)$: a set of all variables used in block *blk*
$IN(blk)$: a set of live variables before *blk*
$OUT(blk)$: a set of live variables after *blk*
$LIVE\_VARS(blk)$: a set of live variables after *blk*
$DEAD\_VARS(blk)$: a set of dead variables after *blk*
2: **end Variables**
3: // Compute live variables at each block of function *func*
4: **procedure** COMPUTE_LIVEVARS(*func*)
5: **for each** block *blk* other than RETURN in function *func* **do**
6: $OUT(blk) = \emptyset$// Initialise OUT set in all blocks
7: **end for**
8: $IN(RETURN) = \{return\ variable\}$
9: $OUT(RETURN) = \{return\ variable\}$
10: **while** Changes to any $IN(blk)$ set **do**// Repeat until fixed-point
11: **for each** block *blk* other than RETURN in backward order **do**
12: // OUT at block *blk* as the union of IN in all its child blocks
13: $OUT(blk) = \bigcup_{s \in succ[blk]} IN(s)$
14: // Compute live variables from last to first code
15: **for each** $c_i \in \{c_n \dots c_0\}$ at block *blk* **do**
16: **if** $c_i == c_n$ **then**
17: // $OUT(blk)$ set is *out* set at last code of block *blk*
18: $out(c_n) = OUT(blk)$
19: **else**
20: // *out* set is *in* set of previous code
21: $out(c_i) = in(c_{i+1})$
22: **end if**
23: // Compute liveness transfer equation
24: $in(c_i) = use(c_i) \cup (out(c_i) - def(c_i))$
25: **end for**
26: $IN(blk) = in(c_0)$// $IN(blk)$ is *in* set at first code of block *blk*
27: **end for**
28: **end while**
29: // Compute live and dead variables at each block
30: **for each** block *blk* other than RETURN **do**
31: $LIVE\_VARS(blk) = OUT(blk)$
32: $DEAD\_VARS(blk) = VARS(blk) - OUT(blk)$
33: **end for**
34: **end procedure**

Our live variable analysis (see Algorithm 4.1) is based on Whiley live variable analysis (Pearce and Groves, 2015b) to compute live variable set before and after each line of code c, denoted by $in(c)$ and $out(c)$, from the last code backward to the first at block *blk*. Suppose we have $c_0, c_1, \ldots, c_n$ at block *blk*. As each block has no branching or interruption (each code has only one child code), we have the liveness transfer equation for code c_i as follows:

$$out(c_i) = in(c_{i+1})$$
$$in(c_i) = use(c_i) \cup (out(c_i) - def(c_i))$$

where $use(c_i)$ is the set of used variables at c_i and $def(c_i)$ represents the set of defined variables at c_i; $in(c_i)$ is the set of live variables before c_i and $out(c_i)$ is the set of live variables after c_i.

The liveness transfer equation can be applied on the composition of all code in a block, so that we can compute *block-level* live variables for each block *blk* in a function by using code-level *in* and *out* sets with below relationship:

$$OUT(blk) = out(c_n)$$
$$IN(blk) = in(c_0)$$

Note `RETURN` block is processed separately as the return variable must be live both before and after the return block.

Procedure 4.2 Live Variable Check

Input: Variable *var* at *code* of function *func*
Output: true: *var* is live after *code* in *func*

```
    // Check var is live after code in func
 1: procedure IS_LIVE(var, code, func)
 2:     if code is a Function Call AND var is used at least once at code then
 3:         return true
 4:     end if
 5:     blk=Locate the block of code at function func
 6:     return (var ∈ LIVE_VARS(blk))? true : false
 7: end procedure
```

Thirdly, we repeat the backward iterative procedure until *IN* sets at all blocks converge, and obtain comprehensive live variable sets such that we can use live variables to determine if a variable is still live at a program point

(see Algorithm 4.2). Furthermore, we can use live variable set to find out dead variables in each block. Because live variable set (Seidl et al., 2012, Chapter 1.7) takes as the union set of variables possible live at least one of child blocks, the complementary set of a block contains only dead variables which are definitely not used at any of child blocks. The dead variable set, denoted by $DEAD\_VARS(b)$, in block *blk* is:

$$DEAD\_VARS(blk) = VARS(blk) - OUT(blk)$$

where $OUT(blk)$ is the live variable set at block *blk* and $VARS(blk)$ contains all the in-use variables at block *blk*.

Dead variable set can be used in bound inference to avoid unstable bounds. As dead variables are not used after a block, their bounds become unpredictable outside their scope. As such, propagating out-of-scoped bounds from dead variables to a block leads to diverged bound changes, and fails converging to the fixed-point and may go into an infinite loop during bound inference phase. To guarantee the termination of bound inference, our bound analyser skips dead variables but combines all possible live variables to produce the bounds for a block.

```
1  function func(int limit) -> int:
2    int i =0
3    int sum=0
4    while i < limit:
5      int j = 0
6      while j < limit:
7        sum = sum + i*j
8        j = j + 1
9      i = i + 1// j becomes dead
10   return sum
```

Listing 4.1: While-loop nest Whiley program

Example 4.1 *The example in Listing 4.1 illustrates a while-loop nest in Whiley. The program uses variable i and j to keep track of the counter at outer and inner loops respectively. We build up the control flow graph for function func and then perform live variable analysis to find live and dead variables in each block, as follows.*

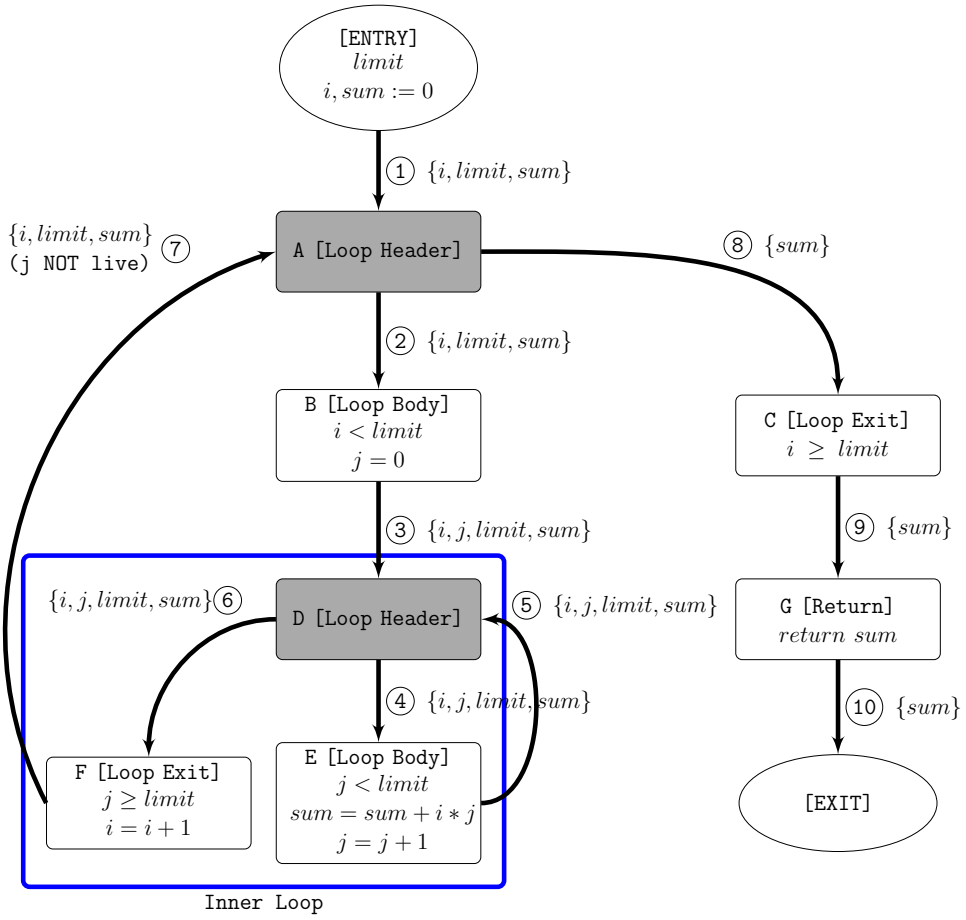


Figure 4.2: Control flow graph of `While-loop nest` program (edge: live variable set)

The *entry* block, as shown in Figure 4.2, stores the values of parameter *limit*, and variable i and *sum*. Then we construct an *outer loop* structure (blocks A, B and C) to place the code at line 4 and 5, and an *inner loop* (blocks D, E and F) to store the code from line 6 to 8. And return block G connects outer loop exit and function *exit* blocks.

The live variable set is shown on the edge after the block. For example, ⑦$\{i, limit, sum\}$ indicates variables i, *limit* and *sum* are used and live after inner loop exit block F. Variable j is used only in the inner loop and becomes dead at the inner loop exit (Block F). So when the analyser propagates bounds from inner loop exit (Block F) to outer loop header (Block A), variable j is skipped to avoid passing out-of-scoped bounds to the inference procedure.

4.1.3 Bound Inference

The bound inference extracts bound constraints from WyIL code and then perform the bound propagation and inference repeatedly until all the bounds are consistent with all the constraints (Malik and Utting, 2005).

Bound Constraint Our bound analyser takes the control flow graph as input, iterates each block in the graph to discover the bound constraints from each line of code at the block, and place the extracted constraints in the corresponding block. By imposing these constraints in each block, we can get a set of bounds that satisfies all the conditions over integer domains and provide possible range of a variable, rather than infinite value.

Table 4.1: Bound constraints and bound propagation rule

WyIL code	**Constraints/Bound Propagation Rule**
`const X = 10`	$(X := 10) \implies d(X) := [10 \dots 10]$
`assign X = Y`	$(X := Y) \implies d(X) := d(Y)$
`ifeq X, Y`	$(X == Y) \implies \begin{cases} d(X) := d(X) \cap d(Y) \\ d(Y) := d(Y) \cap d(X) \end{cases}$
`ifgt X, Y`	$(X > Y) \implies \begin{cases} d(X) := d(X) \cap [min(Y) + 1 \dots \infty] \\ d(Y) := d(Y) \cap [-\infty \dots max(X) - 1] \end{cases}$
`ifge X, Y`	$(X \geq Y) \implies \begin{cases} d(X) := d(X) \cap [min(Y) \dots \infty] \\ d(Y) := d(Y) \cap [-\infty \dots max(X)] \end{cases}$
`iflt X, Y`	$(X < Y) \implies \begin{cases} d(X) := d(X) \cap [-\infty \dots max(Y) - 1] \\ d(Y) := d(Y) \cap [min(X) + 1 \dots \infty] \end{cases}$
`ifle X, Y`	$(X \leq Y) \implies \begin{cases} d(X) := d(X) \cap [-\infty \dots max(Y)] \\ d(Y) := d(Y) \cap [min(X) \dots \infty] \end{cases}$

Definition 4.1 *Bound Definition and Constraints*

Variables X, Y and Z each has a domain d with lower and upper bounds, denoted by $d(X) = [min(X) \ldots max(X)]$ where functions $'min'$ and $'max'$ get the minimal and maximal values of a domain respectively. Domain $d(X)$ indicates the output domain of variable X after being applied with a bound constraint. Each domain is initialised with an empty value $\emptyset$ which represents unknown bounds.

The bound union operator, denoted by $\cup$, produces a new domain that contains two input domains and finds the smaller lower bound and larger upper bound of input domains, as follows.

$$d(X) := d(Y) \cup d(Z)$$

$$\implies d(X) := \begin{cases} d(Y) \ \textit{if} \ d(Z) \ \textit{is} \ \emptyset \\ d(Z) \ \textit{if} \ d(Y) \ \textit{is} \ \emptyset \\ [min(min(Y),\ min(Z)) \ldots max(max(Y),\ max(Z))] \end{cases}$$

The domain intersection operator, denoted by $\cap$, outputs a domain that includes both two input domains and finds the larger lower bound and smaller upper bound of input domains, as follows.

$$d(X) := d(Y) \cap d(Z)$$

$$\implies d(X) := \begin{cases} \emptyset \ \textit{if} \ d(Z) \ \textit{is} \ \emptyset \\ \emptyset \ \textit{if} \ d(Y) \ \textit{is} \ \emptyset \\ [max(min(Y),\ min(Z)) \ldots min(max(Y),\ max(Z))] \end{cases}$$

Each line of WyIL code type could be encoded and expressed in the form of bound constraint and bound propagation rules such that the resulting bounds satisfy all given constraints. Table 4.1 lists the bound rules for an equality, relational and assignment. Our analysis also supports arithmetic operators, i.e. unary negation, addition and multiplication. The propagation rule of a negative operation is to negate the maximal and minimal values and swap

them:

$$d(-X) \implies [-max(X) \ldots - min(X)]$$

For an addition $X := Y + Z$, the bound propagation rule updates domain X with the sum of minimum and maximum of Y and Z, as follows.

$$d(X) := d(Y + Z) := [min(Y) + min(Z) \ldots max(Y) + max(Z)]$$

For instance, domains Y and Z are $[0 \ldots 5]$ and $[-2 \ldots 2]$ respectively, and the resulting domain X is $[-2 \ldots 7]$ and domains Y and Z remain unchanged.

For a multiplication $X := Y \times Z$, the bound rule explores the limits of variable Y and Z, and calculates all the products of maximal and minimal values to find the minimum and maximum of the resulting domain X.

$$d(X) := d(Y \times Z) := \begin{cases} min' = min(min(Y) * min(Z),\ min(Y) * max(Z), \\ \qquad\qquad max(Y) * min(Z),\ max(Y) * max(Z)) \\ max' = max(min(Y) * min(Z),\ min(Y) * max(Z), \\ \qquad\qquad max(Y) * min(Z), max(Y) * max(Z)) \\ d(X) = [min' \ldots max'] \end{cases}$$

Consider the above example. Domain Y is $[0 \ldots 5]$ and domain Z is $[-2 \ldots 2]$. The combination of variable Y multiplied by Z are:

$$d(Y \times Z) := \begin{cases} min(Y) * min(Z) = 0 \\ min(Y) * max(Z) = 0 \\ max(Y) * min(Z) = -10 \\ max(Y) * max(Z) = 10 \end{cases}$$

So the result domain is $d(X) = [-10 \ldots 10]$.

We define constraints and bound propagation rules, and will go through bound inference to infer constraints and bounds for a function.

Procedure 4.3 Tree-Traversal Bound Inference for a Function

Input: Function *func* is a function; Argument Bounds *args* of function *func*.
Output: The domain of return variable *ret* of function *func*

```
1: Variables
2:     blk.d is the domain set of block blk; blk.d(var) is the domain (lower and
   upper bounds) of variable var in block blk.
3: end Variables
4: procedure IS_REACHABLE(blk)// Check the reachability of block blk
5:     return (Any domain ∈ blk.d == ∅) ? false : true
6: end procedure// Return true if blk does not have empty domain
7: // Infer bounds of function func using breath-first or depth-first traversal
8: procedure INFER_BOUNDS(func, args)
9:     cfg = BUILDCFG(func)// Build control flow graph of function func
10:    EXTRACTCONSTRAINTS(cfg)// Extract constraints in each block
11:    INIT(func) // Initialise each domain in each block with ∅
12:    deque.add(cfg.getEntry())// Put entry to deque as starting block
13:    while deque is NOT empty do
14:        blk = deque.poll()// Retrieve block in breath-first or depth-first order
15:        if blk is a function call then// Bound inference on a function call
16:            callee = GETCALLEDFUNCTION(blk)
17:            args = GETARGUMENTBOUNDS(bounds, blk)
18:            // Infer the bounds of called function
19:            ret = INFER_BOUNDS(callee, args)
20:            // Add the domain of variable ret as a constraint to block blk
21:            ADDCONSTRAINT(ret, blk)
22:        end if
23:        // Infer the domains of all variables in block blk
24:        blk.d_in := blk.d// Store domain set of block blk before inference
25:        blk.d := {}
26:        for each parent block in blk do
27:            for each var ∈ parent.vars do
28:                if var is a live variable in parent then
29:                    // Propagate domains of live variables from parents to blk
30:                    blk.d := blk.d ⋃ parent.d(var)
31:                end if
32:            end for
33:        end for// Produce initial value of blk.d from parent blocks
34:        for each constraint ∈ blk.constraints do
35:            Apply the bound propagation rules of constraint on blk.d
36:        end for// Produce blk.d domains consistent with all constraints
37:        if blk.d has any change (blk.d ≠ blk.d_in) then
38:            Add children blocks of blk (except EXIT) to deque
39:        end if// We start inferring the bounds of child blocks
40:    end while// Repeat until the domains of all blocks become stable
41:    // Produce final domains of each variable in function func at EXIT block
42:    exit := ⋃{blk.d • (∀ blk : BLOCKS • is_reachable(blk))}
43:    return exit.d('ret') // Return the domain of return variable ret
44: end procedure
```

Bound Inference *Bound inference* determines the maximal and minimal ranges or a domain of an integer variable that it is used in a function. Once the control flow graph of the function is built up (using procedure `buildCFG`) and the bound constraints are extracted and added to the corresponding blocks in the graph (using procedure `extractConstraints`), the bound analyser starts the bound inference in depth-first or breath-first block order and produces the bounds satisfying constraints in each block. Then the analyser iterates each block and combines the inferred bounds to yield the final domain results for each variable in the function.

Bound Inference on a Function The bound analyser takes the WyIL code of function *func* as input, and outputs the inferred bounds of the function, including return variable *ret*, all local variables and input parameters. The bound inference on a function (see procedure `Infer_Bounds` in Algorithm 4.3) consists of four steps.

Firstly, the analyser goes through every block of the control flow graph and initialises each domain in one block to $\emptyset$ (using procedure `INIT`). Then we use the *deque* data structure to store all the blocks that have bound changes. And entry block is pushed into the deque so that we can start the block inference.

Secondly, the analyser takes out one block from the deque in either depth-first (Last-In First-Out) or Breath-First (First-In First-Out) order and carries out *block bound inference* as follows.

1. Domain depends on variables and blocks. Then, we represent a function $blk.d : VAR \rightarrow domain$ which maps a variable to its domain in block blk. $blk.d(var)$ is the domain of variable var in block blk.

2. $blk.d_{in}$ stores the domain set of block blk before block bound inference.

3. We reset domain set of block blk and take union of every live variable' domain from all the parent blocks of block blk and produce initial block domain set $blk.d$ for bound inference. By doing so we can restrict the variables of block blk to only two conditions:

- The variables are first used in the block *blk* or,
- The variables are live (not dead) in parent block.

These variable rules guarantee every domain is consistent with the block scope that it appears in, and thereby avoid propagating out-of-scope bounds to blocks and causing unstable convergence during inference.

4. We iterate every constraint imposed by the code in block *blk* to infer or propagate the bounds and produce the resulting block domain set $blk.d$, which is satisfied with all constraints in block *blk*.

5. After inferring the bounds of block *blk*, we add *blk*'s child blocks to *deque* for further block inference when the domain set of block b has any change, i.e. $blk.d$ is not the same as $blk.d_{in}$.

6. We proceed to the next block in *deque* and start its block bound inference described as above. This procedure repeats until the deque becomes empty and all bounds converge to the fixed point, at which every bound in the domain set of each block stays unchanged and stable.

Finally, the bound analyser combines the inferred bounds of each block to produce the final resulting domains for all integer variables of function *func*, including return variable, all local variables and passing parameters.

Some blocks may contain empty domains (due to empty intersection on bound inference) and become unreachable, in which case the program flow does not execute the block. Thus, the bound analyser performs reachability check (see `Is_Reachable` procedure in Algorithm 4.3), ignores unreachable blocks and take union of the bounds in remaining ones, in order to approximate the comprehensive domains of integer variables in the execution of function *func*. These resulting domains may be over-estimated but can be used to determine a fixed-width integer type that does not cause arithmetic overflows.

For a function call, we need analysing the domains of passing parameters and return variable described as follows.

Bound Inference for a Function Call The above algorithm 4.3 also shows the bound inference for a function call. The bound analyser passes the bounds of parameters as constraints to the called function, and performs the bound inference on the function, and then propagates the return bounds as a constraint to caller site. We will illustrate the procedure with below example.

Listing 4.2: Whiley program

```
1 function f(int x)->(int r)
2 ensures r >= 0:
3   if x < 10:
4     return 1
5   else:
6     if x > 10:
7       return 2
8   return 0
```

Table 4.2: Bound results

Input Domain	Output Domain
$d(x) := [1 \dots 1]$	$d(r) := [1 \dots 1]$
$d(x) := [10 \dots 10]$	$d(r) := [0 \dots 0]$
$d(x) := [11 \dots 11]$	$d(r) := [2 \dots 2]$

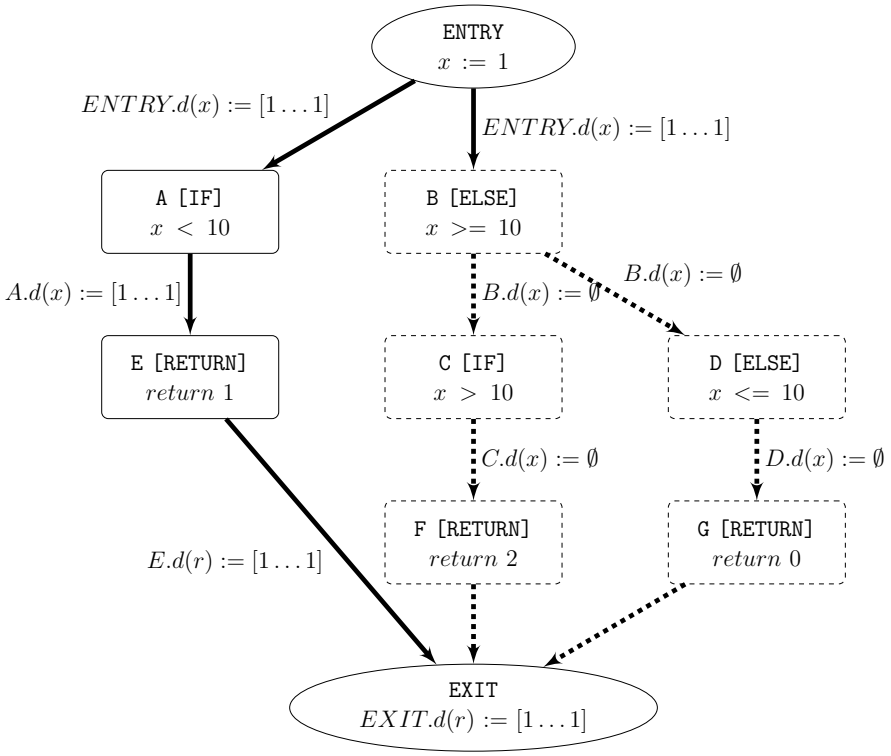


Figure 4.3: Bound inference and reachability check of `If-Else` program with $x := 1$ (solid: reachable, dashed: unreachable)

Example 4.2 *Consider the above example. Function f takes an integer x as input and returns an integer r as output. The input and output domains are listed in Table 4.2. Figure 4.3 shows the bound inference for $ENTRY.D(x) =$*

$[1 \dots 1]$. *Only block A is reachable as block B and others become unreachable due to empty intersection between input bounds and constraints* $B.d(x) := [1 \dots 1] \cap [10 \dots \infty] := \emptyset$.

This example shows that our context-sensitive bound inference procedure can produce the output bounds corresponding to the domain of input parameters. In our example, when encountering a function call, our bound analyser passes the domain of input parameter $ENTRY.d(x) := [1 \dots 1]$ to the called function f and then performs the bound inference in each block of function f. After inferring and converging all the bounds to fix-points, our analyser then checks the reachability of each block in function f and takes the union of bounds at all reachable blocks to yield the output domain of function return $EXIT.d(r) := [1 \dots 1]$.

Our analyser evaluates each individual function call with respect to input parameter and stores the return bounds separately, as shown in Table 4.2. Once all the function calls have been analysed, our analyser combines all the inferred bounds into one domain set. That mean, each domain in the resulting set is the union of bounds of these three calls and large enough to store all the values during calls. As such, using these resulting domains can choose a safe and fixed-width integer types for their associated variables so that arithmetic overflows does not occur in the generated code.

Consider our example again. The final return domain of function f is the union of bounds of all three function calls, i.e. $EXIT.d(r) := [1 \dots 1] \cup [0 \dots 0] \cup [2 \dots 2] := [0 \dots 2]$. With this range, we can use a unsigned 16-bit integers to store the value of return variable in function f.

For a while-loop, our bound inference, described as above, needs to go through all loop iterations to repeatedly estimate the bounds of loop variables and converge to stable domains. When the loop is too large to analyse, our analysis takes too long time to be executed and does not always terminate. Thus, we modify our bound inference procedure with the following widening operator, which is used to accelerate inference time and proven to terminate.

4.1.4 Widening Operator

Abstract interpretation-based widening operator (Cortesi and Zanioli, 2011) is an over-approximation technique to speed up time to the fixed point without executing all loop iterations. In this project, the widening operator can be operated in *naive* or *gradual* mode. The former follows Cousot's original design to jump straight into $\pm\infty$ whilst the latter widens bounds against a list of thresholds.

Procedure 4.4 Bound Inference using Naive Widening Operator

Input: Block *blk*
Output: Return the widen bounds $blk.d_{widen}$ in block *blk*

```
1: Variables
2:     blk.d_in(var) is domain d(var) before a loop iteration of bound inference;
3:     blk.d(var) is domain d(var) after the loop iteration of bound inference;
4:     ub_c(var) is the counter of upper bound for domain d(var);
5:     lb_c(var) is the counter of lower bound for domain d(var).
6: end Variables
   // Check bound changes and widen the bounds with threshold
7: procedure NAIVE_WIDEN_BOUND(blk)
8:     for each var in blk do
9:         // Widen upper bound every subsequent three iterations
10:        if upper(blk.d(var)) > upper(blk.d_in(var)) then
11:            // The upper bound increases in this iteration
12:            ub_c(var) + +
13:            if ub_c(var) == 3 then
14:                // Widen the upper bound of d(var) in block blk to ∞
15:                blk.d(var).upper := +∞
16:                ub_c(var) := 0// Reset upper bound's counter
17:            end if
18:        else
19:            ub_c(var) := 0// Reset upper bound's counter
20:        end if
21:        // Widen lower bound every subsequent three iterations
22:        if lower((blk.d(var)) < lower(blk.d_in(var)) then
23:            // The lower bound decreases in this iteration
24:            lb_c(var) + +
25:            if lb_c(var) == 3 then
26:                // Widen lower bound of d(var) in block blk to -∞
27:                blk.d(var).lower := −∞
28:                lb_c(var) := 0// Reset lower bound's counter
29:            end if
30:        else
31:            lb_c(var) := 0// Reset lower bound's counter
32:        end if
33:    end for
34:    return blk.d// Return the widen domain set
35: end procedure
```

Definition 4.2 *Naive Widening Operator* ∇

$$\emptyset \nabla x = x$$

$$x \nabla \emptyset = x$$

$$[l_n, u_n] \nabla [l_{n+1}, u_{n+1}] = [l', u'], where$$

$$l' = \begin{cases} -\infty, & IF\ l_{n+1} < l_n \\ l_n, & otherwise \end{cases} \quad and\ u' = \begin{cases} \infty, & IF\ u_{n+1} > u_n \\ u_n, & otherwise \end{cases}$$

The naive widening operator ∇ can be used to extrapolate the unstable bounds of an interval to $\pm$ infinity. The naive widening operator ∇ observes the increase of upper bound at each iteration and then decides whether to blow out the upper bound to $+\infty$. In the same manner, the operator converges decreasing lower bounds to $-\infty$. Within finite steps, the widening operator can stabilise the bounds and accelerate the time of bound inference.

Algorithm 4.4 shows that, in each loop iteration the naive widening operator checks the bound changes of each variable and keeps track of its number of changes, to determines whether the upper or lower bound widens to $\pm\infty$. If so, we have ultimately stationary bounds to enforce termination of the loop and to stabilise the bounds within finite and fewer iterations. Therefore, the convergence time of bound inference can be accelerated.

The naive widening operator throws away bound information generously and thus may over-approximate the bounds ($\pm\infty$), and reach the bound convergence earlier than expected. To use widen operator more wisely, we introduce three widen parameterisation:

- Block traversal order can be specified to infer the blocks in breath-first or depth-first order.

- Feedback block set (Seidl et al., 2012) is used to restrict the widening operation is only applied on loop header blocks, rather than on every block, so that we can reduce the number of bound checking on widen operator and improve the efficiency.

- Strict widening rule is applied to limit the widen operator is used every subsequent three iterations. We observe the bound change and count the number of iterations and reset the counter if any bound stays unchanged or does not increase or decrease continuously.

Table 4.3: Threshold values of fixed-width integer type

Threshold	**Description**	**Value**
$INT64_{max}$	`max`(signed `int64_t`)	$2^{63} - 1$
$INT32_{max}$	`max`(singed `int32_t`)	$2^{31} - 1$
$INT16_{max}$	`max`(singed `int16_t`)	$2^{15} - 1$
$INT16_{min}$	`min`(singed `int16_t`)	-2^{15}
$INT32_{min}$	`min`(singed `int32_t`)	-2^{31}
$INT64_{min}$	`min`(singed `int64_t`)	-2^{63}

Widening with thresholds (Blanchet et al., 2003) can improve the precision of interval analysis and proves the boundedness of variables by using a series of thresholds defined in C99 `stdint.h` header file (see Table 4.3).

Definition 4.3 *Threshold Set*
A maximal threshold set TH_{max} is a set which contains all maximal values of integer types in ascending order, i.e.

$$TH_{max} = \{INT16_{max},\ INT32_{max},\ INT64_{max},\ +\infty\}$$

A minimal threshold set TH_{min} is a set which contains all minimal values of integer types in ascending order, i.e.

$$TH_{min} = \{INT16_{min},\ INT32_{min},\ INT64_{min},\ -\infty\}$$

The gradual widening operator $\overline{\nabla}$ goes through each threshold to find a suitable interval which can stabilise the bounds to reach the fixed point.

Definition 4.4 *Gradual Widening Operator* $\overline{\nabla}$

$$\emptyset \overline{\nabla} x = x$$

$$x \overline{\nabla} \emptyset = x$$

$$[l_n, u_n] \,\overline{\nabla}\, [l_{n+1}, u_{n+1}] = [l^{th}, u^{th}], where$$

$$l^{th} = \begin{cases} \max\{th_{min} \in TH_{min} \bullet (th_{min} < l_{n+1})\}, & if\ l_{n+1} < l_n \\ l_n, & otherwise \end{cases}$$

$$u^{th} = \begin{cases} \min\{th_{max} \in TH_{max} \bullet (th_{max} > u_{n+1})\}, & if\ u_{n+1} > u_n \\ u_n, & otherwise \end{cases}$$

The gradual widening operator broadens an increasing upper bound to the minimum of maximal thresholds until the bound stays unchanged. In the same manner, the operator widens decreasing lower bound to the maximum of minimal thresholds. The resulting bounds can provide the code generator to choose a proper fixed-sized data type for integer variable such that the inferred bound falls within the range, e.g. the bound between `INT16_MAX` and `INT16_MIN` can be stored with an *int16_t* integer.

```
1 function f(int limit)-> int
2 requires limit < 1000000:
3   int i = 0
4   int sum = 0
5   while i < limit:
6     sum = sum + i
7     i = i + 1
8   return sum
```

Listing 4.3: `While-loop` Whiley Program

Example 4.3 *Consider the above while-loop Whiley Program to compute the total of integer values from 0 to the loop bound which is the passed parameter of function f .*

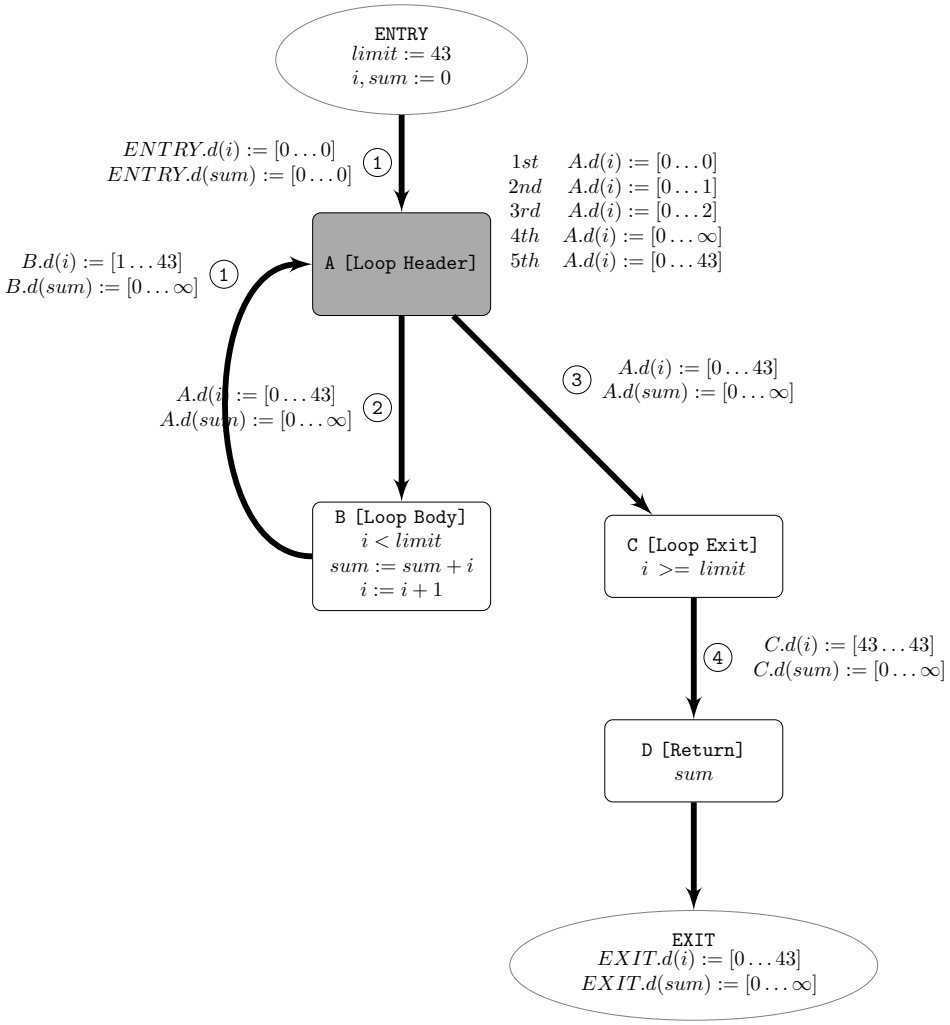


Figure 4.4: Control flow graph of `While-loop` program using naive widen operator in breath-first order

This example uses an incremental while-loop to calculate the summation of a given limit (43). As shown in Figure 4.4, the program is broken down into several blocks and a loop structure (A: loop header, B: loop body and C: loop exit). All the directed edges show the relations among blocks and the circled number indicates the sequence block order on bound inference.

The analyser iterates each block in the breath-first order and infers the bound in each individual block. And the widening operator is applied only on loop header to improve its efficiency and accelerate the bound convergence.

Table 4.4: Bound results using naive widening operator in breath-first order (*limit*:=43, *l*: lower bound, *u*: upper bound)

Iteration	Block	Entry		A		B		C		G		Exit	
	VARS	l	u	l	u	l	u	l	u	l	u	l	u
0	i	0	0	0	0	1	1	∅	∅				
	sum	0	0	0	0	0	0	0	0				
1	i			0	1	1	2	∅	∅				
	sum			0	0	0	1	0	0				
2	i			0	2	1	3	∅	∅				
	sum			0	1	0	3	0	1				
3	i			0	∞	1	43	43	∞	43	∞		
	sum			0	3	0	45	0	3	0	3		
4	i			0	43	1	43	43	43	43	43	0	43
	sum			0	∞	0	∞	0	∞	0	∞	0	∞

Table 4.5: Bound results using naive widening operator in depth-first order (limit:=43, l: lower bound, u: upper bound)

Iter.	Block	Entry		A		C		G		B		Exit	
	VARS	l	u	l	u	l	u	l	u	l	u	l	u
0	i	0	0	0	0	∅	∅			1	1		
	sum	0	0	0	0	0	0			0	0		
1	i			0	1	∅	∅			1	2		
	sum			0	0	0	0			0	1		
2	i			0	2	∅	∅			1	3		
	sum			0	1	0	1			0	1		
3	i			0	∞	43	∞	43	∞	1	43		
	sum			0	3	0	3	0	3	0	45		
4	i			0	43	43	43	43	43	1	43	0	43
	sum			0	∞	0	∞	0	∞	0	∞	0	∞

Table 4.4 and 4.5 are the steps of bound inference coupling with the naive widening operator in breath-first and depth-first block order respectively. Re-

sults show that the widen operator can reduce 43 fixed-point iterations down to 5.

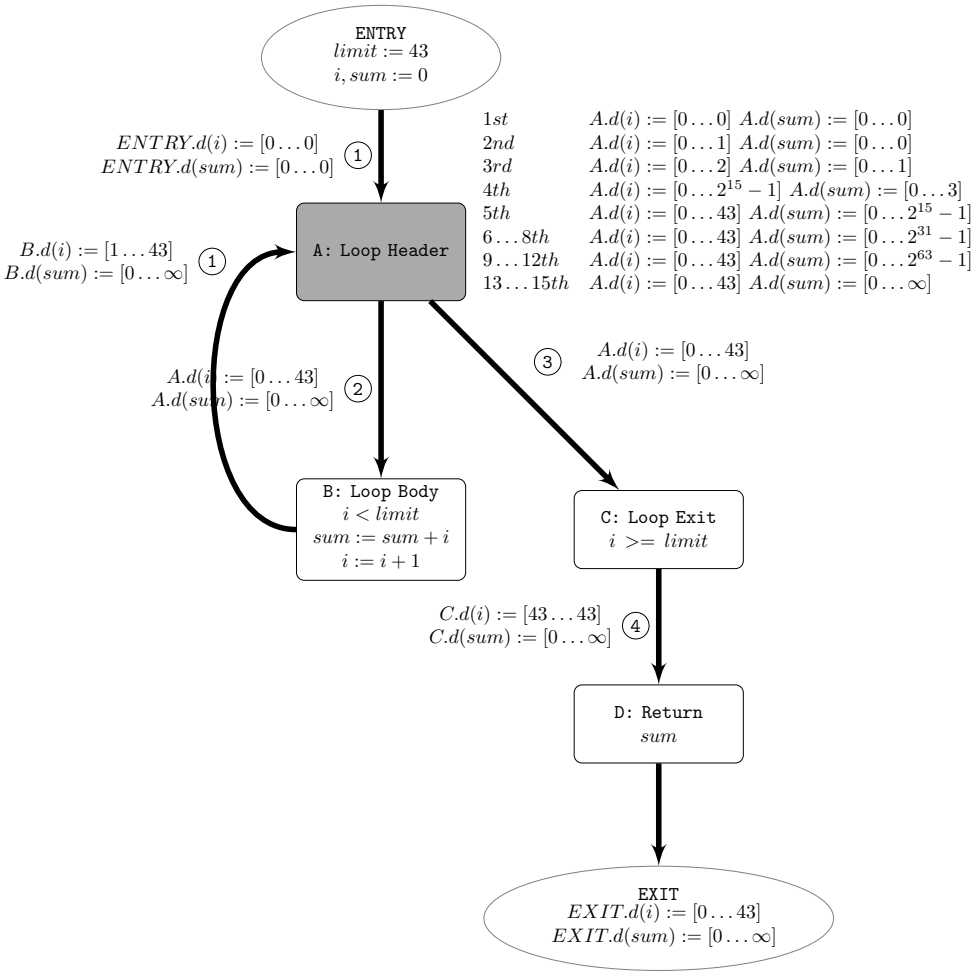


Figure 4.5: Control flow graph of `While-loop` program using gradual widen operator in breath-first traversal

Table 4.5 applies the gradual widen operator on variable i and jump to the range of 16 bit integer $2^{15} - 1$ and converges to limit on loop body and then reach fixed-point (see 4th and 5th iterations). However, due to the lack of constraints on variable *sum*, the widen operator needs to be re-applied and increases the upper bound from 16, 32, 64-bit integer up to positive ∞ to terminate the bound inference.

Even though the resulting bounds in this example do not make any difference, the gradual widen operator (Blanchet et al., 2003) can give more precise results on more complex loop update, such as $x = x/2 + 100$.

Example 4.4 *Function find uses a while-loop with break statement as follows.*

```
1 function find(int limit, int item) -> int:
2   int r=0
3   while r<limit where 0<=r:
4     if r == item:
5       break
6     r=r+1
7   return r
```

Listing 4.4: While-loop with break source Whiley program

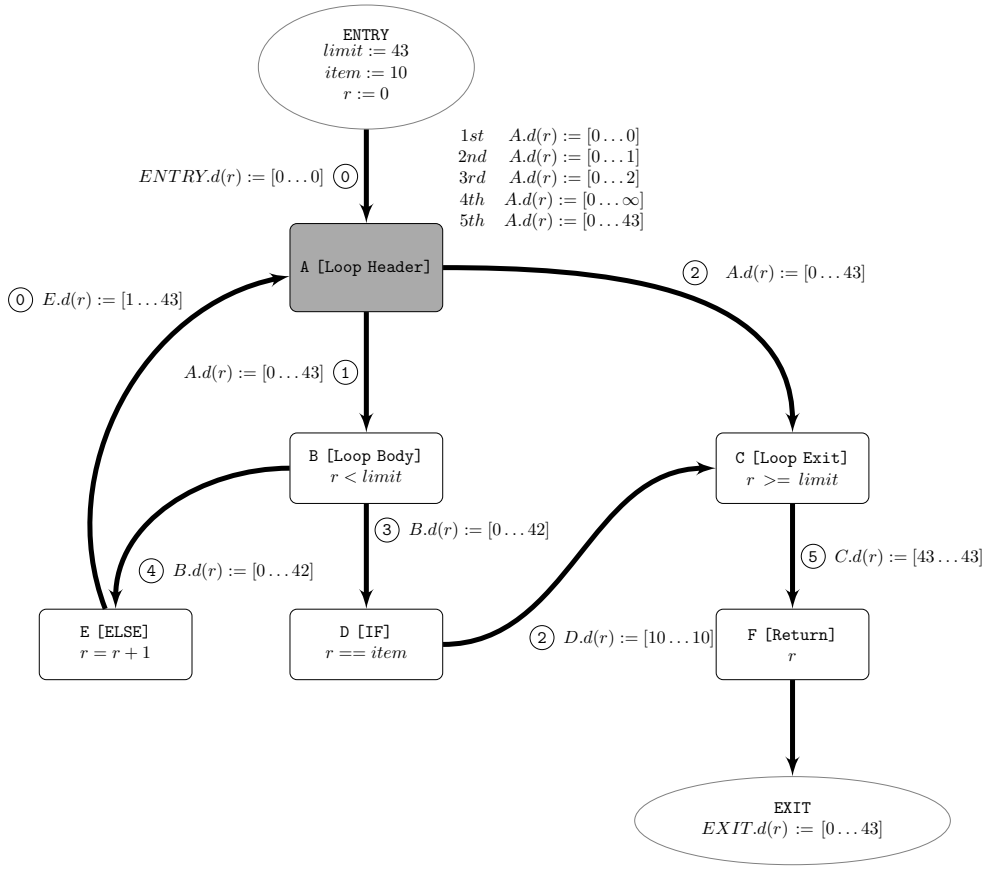


Figure 4.6: Control flow graph of `While-loop with break` program using naive widening Operator in breath-first traversal

The bound inference in breath-first order is as follows:

- Variable r at block A is widened to ∞ after three visits and then reaches the fixed-point $[0 \dots 43]$.
- Variable r is narrowed down to $[10 \dots 10]$ because of equality constraint at D block. Finally, we take union of bounds from block A and D and produce a larger domain $[10 \dots 10] \bigcup [0 \dots 43] = [0 \dots 43]$ at block C.

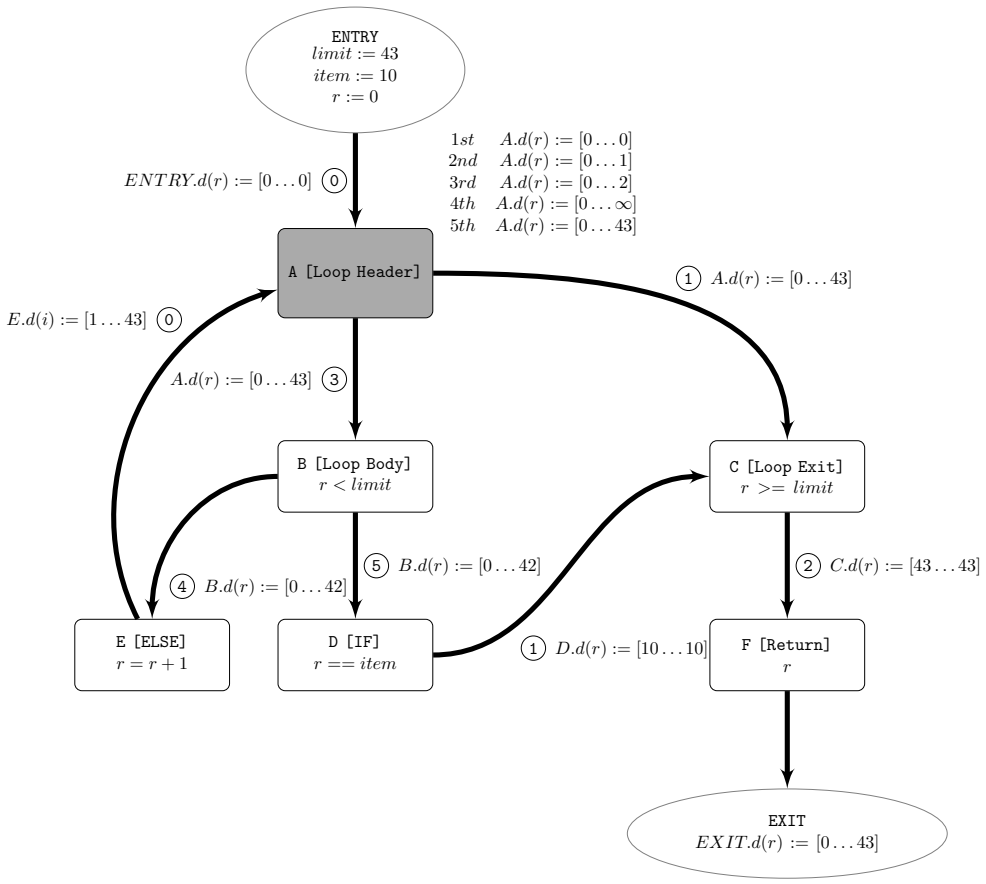


Figure 4.7: Control flow graph of `While-loop with break` program using naive widening operator in depth-first traversal

The depth-first Bound inference produce the same bounds as breath-first.

```
1  function f(int limit) -> int:
2    int i =0
3    int sum=0
4    while i < limit:
5      int j = 0
6      while j < limit:
7        sum = sum + i*j
8        j = j + 1
9      i = i + 1
10   return sum
```

Listing 4.5: Nested While-loop Source Whiley Program

Example 4.5 *Consider a nested while-loop Whiley Program. The outer and inner loop variables are variable i and j respectively. Both of loop bounds are the same (limit).*

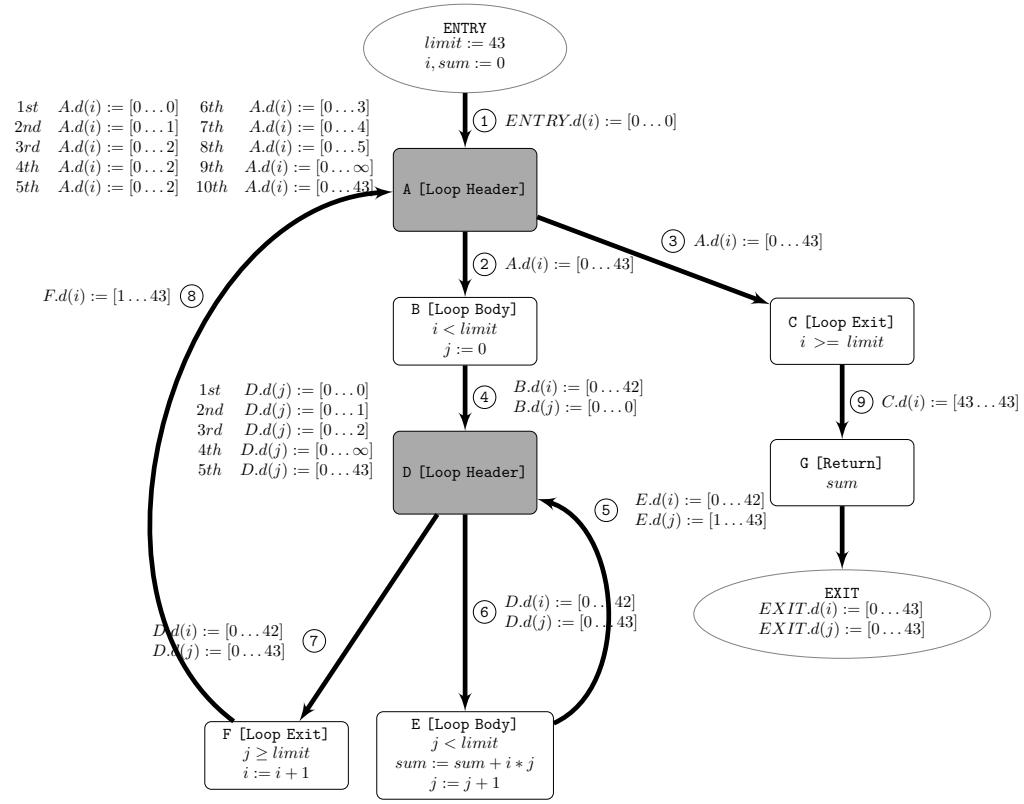


Figure 4.8: Control flow graph of `While-loop nest` program using naive widening operator in breath-first traversal

Breath-First Bound inference The bound inference in breath-first order explores all the sibling blocks first and then move on to the next level, so the block orders are:

$$A, B, C, D, E, F, G$$

The figure shows that

- Variable j increases its upper bound with three visits in D block, and by applying widen operator, converges the domain to fix interval $[0 \ldots 43]$.
- Variable *sum* also increases consecutively inside the inner loop header during 6 to 8 visits at D block, so blow out the bound to ∞.
- Variable i stays at $[0 \ldots 2]$ for the first few visits. But once variable j and *sum* reach a fixed-point, variable i start changing its value, widen the bound and reach the fixed point $[0 \ldots 43]$.

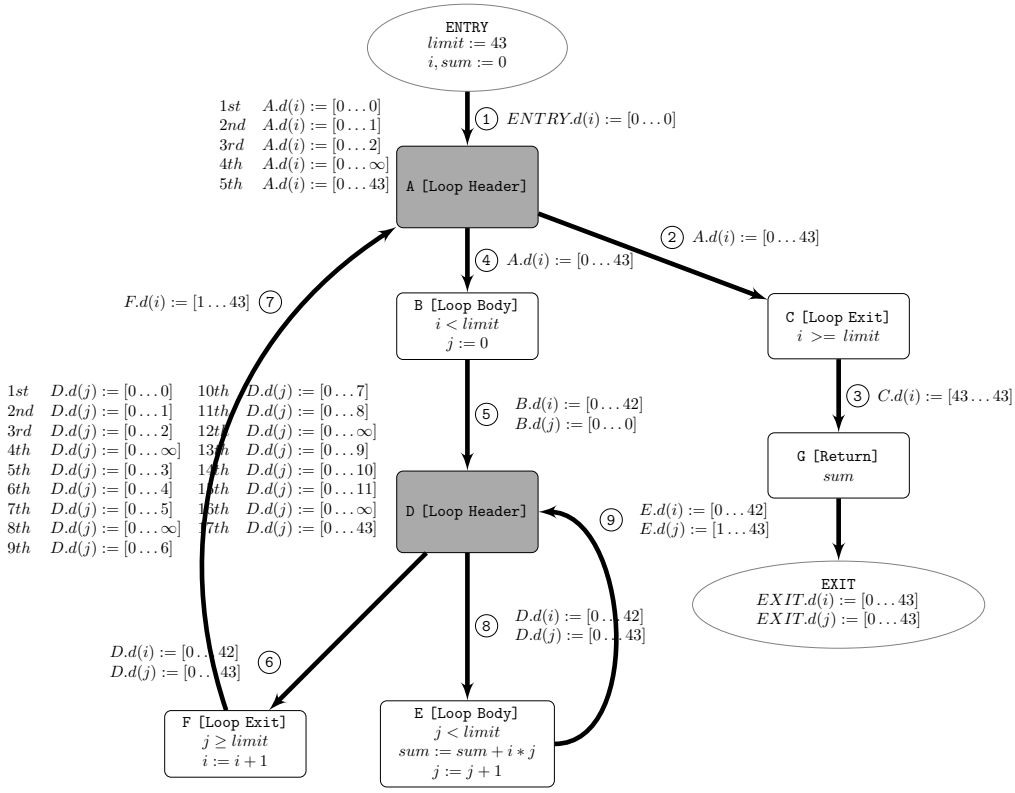


Figure 4.9: Control flow graph of `While-loop nest` program using naive widen operator in depth-first traversal

Depth-First Bound inference The depth-first search traverses blocks at the deepest level and then back-traces the sibling block, so the orders are as below:

$$A, C, G, B, D, F, E$$

The analyser goes through the inner loop (D, F, E blocks) and blows out the upper bound of variable j to ∞ until F block becomes reachable. So the analyser can go back block A and infer the bound in outer loop. The figure shows that:

- variable i is blown out to ∞ once and then yields bounded domain $[0 \ldots 43]$ during the first 5 visits in block A.
- Every visit in block A recurs the bound inference on the inner loop. That means it will go through blocks D, F and E until domain j is large enough to make F block reachable so it will proceed to outer loop.

Variable j is widened to ∞ every three visits in D block, but due to the memorised domain from previous visit in E block, variable j is then reset to the fixed interval. For example, domain j is reset to $[0 \dots 3]$ at 5th visit in block D.

In round-robin iterations, domain j is repeatedly widened to ∞ whereas variable *sum* stays at unbounded domain $[0 \dots \infty]$ after applying widen operator.

- At the last visit at block D, variable j propagates the bound from blocks E and B, and produce the fixed interval $[0 \dots 43]$. As domain i, j and *sum* all reach the fixed-point and do not change the bound in any block, the bound inference procedure terminates.

4.2 Pattern Matching and Transform

In this section we show how our analyser finds the pattern of a function and, if matched, performs pattern transformation to improve the efficiency of resulting code.

4.2.1 Pattern

Our analyser has been built in with several patterns, including *while-loop*, *while-loop increment*, *while-loop decrement* and *append array* patterns.

Definition 4.5 *(Symbol Set) Let VARS be a set of symbols (variables and values).*

Definition 4.6 *(While-Loop Pattern) Function func is said to satisfy a while-loop pattern if func contains a while-loop structure, where the loop variable V initialises to INIT value, and the loop condition has a loop comparator OP and loop bound B.*

The form of while-loop pattern is:

```
⟨V⟩ = ⟨INIT⟩; // Initialise loop variable V
while ⟨V⟩ ⟨OP⟩ ⟨B⟩: // Loop condition
    ⟨BODY not assigning to/updating V and not changing B⟩
    ⟨Update V⟩
```

where variable V keeps track of the loop counter; expression $INIT$ denotes the initial value of loop variable V; OP is the comparing operator of loop condition; expression B denotes the loop bound; $BODY$ represents a sequence of code inside loop body and does not update loop variable V nor loop bound B. Note expression $INIT$ and B do not contain or update loop variable V.

A *while-loop* can be categorised as either an incremental or decremental while-loop pattern by the value of loop update. With the information of loop update and loop bound, we can estimate the number of loop iterations $loop\_iters(V)$ described as follows.

Definition 4.7 *(Incremental While-loop Pattern) Function func is said to satisfy a while-loop increment loop if func is matched with while-loop pattern and the loop variable V is incremented by one in each iteration. The form of incremental while-loop pattern is:*

```
⟨V⟩ = ⟨INIT⟩ // Initialise loop variable
while ⟨V⟩ ⟨OP⟩ ⟨B⟩:// Loop condition
    ⟨BODY not updating V or B⟩
    ⟨V⟩ = ⟨V⟩ + 1 // Loop variable must be increased by one
```

where OP can only be $<$ or $\leq$; expression B and $INIT$ are taken before entering the loop. The number of loop iterations $loop\_iters(V)$ is

$$loop\_iters(V) = \begin{cases} B - INIT, & OP \text{ is } < \\ B - INIT + 1, & OP \text{ is } \leq \end{cases}$$

Definition 4.8 *(Decremental While-loop Pattern) Function func is said to satisfy a while-loop decremental loop if func is matched with while-loop pattern*

and the loop variable is decremented by one in each iteration. The form of decremental while-loop pattern is:

```
⟨V⟩ = ⟨INIT⟩  // Initialise loop variable
while ⟨V⟩ ⟨OP⟩ ⟨B⟩:// Loop condition
    ⟨BODY not updating V or B⟩
    ⟨V⟩ = ⟨V⟩ − 1  // Loop variable must be decreased by one
```

where OP can only be $>$ *or* $\geq$*; expression B and INIT are taken before entering the loop. The number of loop iterations loop_iters*(V) *is*

$$loop\_iters(V) = \begin{cases} INIT - B, & OP\ is\ > \\ INIT - B + 1, & OP\ is\ \geq \end{cases}$$

In some cases, the while-loop can be used to build up and extend an array dynamically to accommodate new array items by using function *append*. Function *append* is a standard system library function and makes a copy of input array, puts a new item into its last and returns the new array, as shown in the following Whiley program.

```
1  // Copy array 'items' to 'nitems' and append 'item' to 'nitems'
2  function append(byte[] items, byte item) -> (byte[] nitems)
3    ensures |nitems| == |items| + 1:
4    nitems = [0b; |items|+1]//Create an array filled in 0 (length: |items|+1)
5    int i = 0
6    while i < |items|:
7      nitems[i] = items[i]
8      i = i + 1
9    nitems[i] = item
10   // Return the new array
11   return nitems
```

Listing 4.6: Function *append* Whiley program

The above loop appends a fixed number of items (1 or more items) to the array every iteration. In this case, there is a linear relation between the number of array elements and the loop iterations. As such, we can express the length of such an array in terms of the number of loop iterations executed.

We propose *append array* pattern to identify such an array manipulation which calls function *append* to add one item to the array within a while-loop, and to give an estimate of the array size before the loop executed, so that we can allocate the necessary memory space for the target array.

Definition 4.9 *(Append Array Pattern) Function func is said to satisfy an append array pattern if func matches with incremental or decremental while-loop pattern, as well as an output array variable ARR. Also, function append is called to add one item to array ARR per loop iteration.*

The form of append array pattern with incremental while-loop is:

```
⟨ARR⟩ = [⟨X⟩; 0]  // Initialize ARR with an empty array
⟨V⟩ = ⟨INIT⟩
while ⟨V⟩ ⟨OP⟩ ⟨B⟩:
    ⟨S_0 not updating V, B or ARR⟩
    ⟨V⟩ = ⟨V⟩ + 1  // Increment loop variable by one
    ⟨S_1 not updating V, B or ARR⟩
    ⟨ARR⟩ = append(⟨ARR⟩, ⟨item_1⟩)  // Append item_1 to array ARR
    ...
    ⟨S_n not updating V, B or ARR⟩
    ⟨ARR⟩ = append(⟨ARR⟩, ⟨item_n⟩)// Append item_n to array ARR
    ⟨S_n+1 not updating V, B or ARR⟩
```

Or the form of append array pattern with decremental while-loop is

```
⟨ARR⟩ = [⟨X⟩; 0]  // Initialize ARR with an empty array
⟨V⟩ = ⟨INIT⟩
while ⟨V⟩ ⟨OP⟩ ⟨B⟩:
    ⟨S_0 not updating V, B or ARR⟩
    ⟨V⟩ = ⟨V⟩ − 1
    ⟨S_1 not updating V, B or ARR⟩
    ⟨ARR⟩ = append(⟨ARR⟩, ⟨item_1⟩)// Append item_1 to array ARR
    ...
    ⟨S_n not updating V, B or ARR⟩
    ⟨ARR⟩ = append(⟨ARR⟩, ⟨item_n⟩)// Append item_n to array ARR
    ⟨S_n+1 not updating V, B or ARR⟩
```

ARR denotes the loop variable. $S_0, S_1, \ldots S_{n+1}$ *each represents some statements that do not contain or update loop variable* V *or array variable ARR.* $item_1, \ldots item_n$ *denotes an array item that is appended to the last.*

In each loop iteration function append is being called n times to append

n items to array ARR. Thus, array ARR grows linearly with the number of function append calls and the number of loop iterations. And we can estimate the size of array ARR, denoted by arr_capacity(ARR):

$$arr\_size(ARR) = loop\_iters(V) \times n$$

where n is the number of function append executed in a loop iteration and loop_iters(V) represents the number of loop iterations.

With above definition, we can use *append array* pattern to pre-allocate the array with an estimate of array size before the execution, so that we can avoid slow array appending but use efficient array update to improve the program efficiency.

Definition 4.10 *(Null Pattern) Function func is said to be a null pattern if func is not matched with any while-loop pattern.*

The pattern matching procedure is straight-forward and described as follows. Given a function, the pattern matcher attempts to iterate each of our patterns and construct the pattern with the code of function. If the pattern can be built up successfully, then the function is matched with the pattern.

As our patterns is inherited from while-loop, we can conduct the procedure hierarchically. That is, we start with the while-loop pattern first and check the function matches it. If so, then we can move on to incremental or decremental while-loop, and even array append until we find the pattern at the deepest level. If no pattern is found, then *NULL* pattern is returned.

4.2.2 Pattern Transformation

Our analyser matches the function with *append array* pattern, and then can perform the code transformation on that function to make use of *preallocated array* pattern and improve the efficiency of program execution.

Definition 4.11 *(From Append Array Pattern to Preallocate Array Pattern) Append array pattern adds n items to the array by using function append in*

each loop iteration, but introduce expensive overheads of array copying. But the append array pattern can be transformed into preallocate array pattern with estimated array size:

$$arr\_size(ARR) = loop\_iters(V) * n = \begin{cases} (B - INIT) \times n, & OP \text{ is } < \\ (B - INIT + 1) \times n, & OP \text{ is } \leq \end{cases}$$

(see append array pattern 4.9 and incremental while-loop pattern 4.7).

Append array pattern	$\Longrightarrow$ ***Preallocate array pattern***
$\langle ARR \rangle = [\langle X \rangle; 0]$	$\langle ARR \rangle = [\langle X \rangle; arr\_size(ARR)]$
$\langle V \rangle = \langle INIT \rangle$	$\langle V \rangle = \langle INIT \rangle$
while $\langle V \rangle$ $\langle OP \rangle$ $\langle B \rangle$:	**while** $\langle V \rangle$ $\langle OP \rangle$ $\langle B \rangle$:
$\langle S_0$ `not updating V, B or ARR`$\rangle$	$\langle S_0$ `not updating V, B or ARR`$\rangle$
$\langle V \rangle = \langle V \rangle + 1$	$\langle V \rangle = \langle V \rangle + 1$
$\langle S_1$ `not updating V, B or ARR`$\rangle$	$\langle S_1$ `not updating V, B or ARR`$\rangle$
$\langle ARR \rangle = append(\langle ARR \rangle, \langle item_1 \rangle)$	$\langle ARR[size] \rangle = \langle item_1 \rangle$
	$size = size + 1$
...	...
$\langle S_n$ `not updating V, B or ARR`$\rangle$	$\langle S_n$ `not updating V, B or ARR`$\rangle$
$\langle ARR \rangle = append(\langle ARR \rangle, \langle item_n \rangle)$	$\langle ARR[size] \rangle = \langle item_n \rangle$
	$size = size + 1$
$\langle S_{n+1}$ `not updating V, B, ARR`$\rangle$	$\langle S_{n+1}$ `not updating V, B, ARR`$\rangle$

Array variable is ARR and loop variable is V; expression X represents the initial value of array item; expression $arr\_size(ARR)$ denotes the estimated size of array ARR; expression INIT is the initial value of loop variable V; OP stands for the comparing operator of loop condition; B is the loop bound; $S_{1...n+1}$ each represents a sequence of code which does not assign to/update loop variable V, loop bound B or array variable ARR; $item_{1...n+1}$ each is the array item; variable size keeps track of the size of array ARR.

Preallocate array pattern uses the estimate of array size to allocate all the necessary space in memory for array *VAR* before loop executed, so that expensive array copying can be replaced with fast and constant-time array update. The

performance of resulting code therefore can be improved. We will illustrate the pattern transformation with the below example.

```
1  // Append an array one by one at each iteration
2  function f(byte[] input) -> (byte[] output):
3    int pos = 0
4    output = [0b;0] // Empty output array
5    while pos < |input|:// Iterate each byte in 'input' array
6      byte index = Int.toUnsignedByte(pos)
7      byte item = input[pos]
8      pos = pos + 1
9      // Append index and item to 'output' array
10     output = append(output, index)
11     output = append(output, item)
12   return output
```

Listing 4.7: Append array Whiley program

Example 4.6 *Consider the above example in Listing 4.7. Suppose variable input is a byte array. Function f takes it as input and produces an array output. The function starts with an empty array and uses function append to copy the output array and add a new item onto the end of array.*

The pattern transformation has two main steps: estimating array size and transforming the code.

Array Size Estimation We firstly find the pattern of function f and then obtain the array size information to perform pattern transformation. Since function f is matched with incremental while-loop, we can know the number of loop iterations is the length of array *input* (see Definition4.7):

$$loop\_iters(pos) = |input| - 0 = |input|$$

Function f is further matched with append array pattern. As the loop makes two *append* function calls every iteration, we can estimate the size of array *output* (see append array pattern 4.9):

$$arr\_size(output) = loop\_iters(pos) \times 2 = |input| \times 2$$

With above information, we can allocate array *output* with double the size of array *input* before the loop. Then inside the loop, we gradually update array *output* with items and count its array size. Finally, outside the loop we then have array *output* filled up with all the items.

Code Transformation According to pattern transformation in Definition 4.11, we can change function f to the following program:

```
1  // Function 'f' uses resize array pattern
2  function f(byte[] input) -> (byte[] output):
3    int pos = 0
4    // Pre−allocate output array with 2x input array size
5    output = [0b;2*|input|]
6    int size = 0// Actual array size
7    while pos < |input|: // Iterate each byte in 'input' array
8      byte index = Int.toUnsignedByte(pos)
9      byte item = input[pos]
10     output[size] = index// Fill in the array with in−place update
11     size = size + 1
12     output[size] = item
13     size = size + 1
14     pos = pos + 1
15   output = resize(output, size)// Resize output array to actual size
16   return output
```

Listing 4.8: Tranformed Function f using Resize Array Pattern

Listing 4.8 shows that array *output* is pre-allocated with the size large enough to hold all its items, so that any out-of-bound array error can be avoided during loop iterations executed. And we use fast array update, instead of slow array append, to populate array *output*. And at the end of function, we reduce array *output* to precise-sized one to save the memory space.

Time Complexity One may be interested in the efficiency improvement obtained from our pattern transformation. Assume the array size is n. The complexity of performing append array pattern is calculated as below.

- Function *append* has a linear-time complexity $O(n)$.
- Function *append* is repeatedly invoked within a loop, so the total number of function calls is the same as loop iterations or n.

So in the worse case the array append pattern is quadratic-timed complexity $O(n * n) = O(n^2)$. However, the preallocate array pattern utilises in-place array update and thus has linear-time complexity $O(n)$. Therefore, we can conclude preallocate array pattern is more efficient than append array.

Chapter 5

Copy Elimination Analysis

Our project (Weng et al., 2017) develops several function analyses, copy elimination analysis and de-allocation analysis to extract the properties of each WyIL code, and then assist our code generator to apply code optimisation and produce efficient code.

5.1 Function Analyses

The function analysers all employ a conservative strategy to extract variable information from functions, and store that information in order to support the copy and de-allocation analysers to make safe code optimisation, while improving the efficiency.

Each function analyser traverses all the functions and processes specific information. Our project includes three 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.

5.1.1 Read-Write Analyser

Procedure 5.1 Read-Write Analysis

Input: *WyIL* file, compiled by Whiley compiler
Output: MUT maps each function to a mutable set

```
    // Collect mutable sets in all functions
 1: procedure MUTABLE_ANALYSIS(WyIL)
 2:     MUT = ∅
 3:     for each func function in WyIL do
 4:         MUT(func) = ∅
 5:         for each code in func do
 6:             lhs ← Extract LHS variable at code
 7:             if lhs != NULL then
 8:                 MUT(f) = MUT(f) ∪ lhs
 9:             end if
10:         end for
11:     end for
12: end procedure
```

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 Procedure 5.1). The variable at right-hand side (RHS) is usually not mutable, because it is copied before update. As we shall see later, 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.

Procedure 5.2 Mutable Check

Input: Variable *var* in function *func*
Output: Return true if *var* is mutated inside *func* function

```
1: procedure isMUTATED(var, func)
2:     return var ∈ MUT(func)
3: end procedure
```

Our read-write analysis conservatively keeps all 'definite' and 'may-be' mutable variables. The check (see Procedure 5.2) weakly identifies a mutable variable, but can strongly detect immutable or read-only ones. This information about read-only variables is used by the copy analyser to decide whether copying is necessary or not.

5.1.2 Return Analysis

Procedure 5.3 Return Analysis

Input: *WyIL* file, compiled by Whiley compiler
Output: RET maps each function to a return set

```
    // Collect return sets for all functions
 1: procedure RETURN_ANALYSIS(WyIL)
 2:     RET = ∅
 3:     for each func function in WyIL do
 4:         RET(f) = ∅
 5:         for each code in func do
 6:             if code is Return then
 7:                 ret ← Extract return variable from code
 8:                 if ret is NOT NULL then
 9:                     RET(f) = RET(f) ∪ ret
10:                 end if
11:             end if
12:         end for
13:     end for
14: end procedure
```

The return analyser (see Procedure 5.3) includes all definite and possible return variables, even those within if-else. The return variable information allows the update from copy analyser to add 'may-be' or aliased return variable after copy removal.

Procedure 5.4 Return Check

Input: Variable *var* at function *func*
Output: Return true if *var* is returned by function *func*

```
    // Check var is returned by function func
1: procedure ISRETURNED(var, func)
2:     return var ∈ RET(func)
3: end procedure
```

Due to the expansion of return set, the return check (see Algorithm 5.4) can be used to effectively detect those non-returnable variables that are never returned by the function, which can allow that memory to be de-allocated 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 de-allocation. Despite the potential memory leak problem, the conservative false alarm can reduce the chances of invalid freeing while maintaining memory safety.

5.1.3 Live Variable Analysis

Procedure 5.5 Liveness Check

Input: Variable *var* at *code* in Function *func*
Output: true: *var* is live after *code* in *func*

```
1: procedure IS_LIVE(var, code, func)
2:     if code is a Function Call AND var is used more than once at code then
3:         return true
4:     end if
5:     blk ← Locate the block of code in function func
6:     return (var ∈ LIVE_VARS(blk))
7: end procedure
```

Live variable analysis (see Algorithm 4.1 in Section 4.1.2) is used to determine whether a variable is still live or used after a specific code (see Procedure 5.5).

Apart from live variable sets, we introduce an extra rule to determine the liveness of a function call parameter when it is used more than once at a call. Consider the function call $func(a,\ a)$. Variable a is used twice at *func* call, so the first parameter a should be considered a live variable at the call because it is passed to the function as second formal parameter.

5.2 Copy Elimination Analysis

The copy elimination analysis (Weng et al., 2017) aims to reduce the number of array copies in generated code whilst avoiding sides effects. Rather than the abstraction-based method (Schnorf et al., 1993) that gives promising results but has difficult limits on implementations, we develop a straightforward analysis tool, similar to alias annotation analysis in Java (Aldrich et al., 2002), to work at intermediate level of Whiley code and to detect where and what copies are unneeded using our live variable analysis, which is based on the live variable analysis in Whiley compiler with variation.

Table 5.1: Copy elimination rule

Function Call $a := f(b)$					
f Mutates b?		F	F	T('maybe')	T('maybe')
f Returns b?		F	T('maybe')	T('maybe')	F
b is live?	F	No Copy	No Copy	No Copy	No Copy
	T	No Copy	Copy	Copy	Copy
No Copy: avoid the copy and pass b to called function f					

Procedure 5.6 Copy Elimination Check

Input: Variable *var* at *code* of function *func*
Output: Return true if copy of *var* can be removed at *code* of function *func*

```
1: Variables
   LiveAnalyser: live variable analyser, ReadWriteAnalysis: Read-Write analyser, ReturnAnalysis: Return analyser
2: end Variables
3: procedure IS_COPYELIMINATED(var, code, func)
4:     if var is array type then
5:         isLive ←liveAnalyser.ISLIVE(var, code, func)
6:         if ¬isLive then// var is NOT live at code at caller
7:             return true // Copy can be removed
8:         end if
9:         if code is a function call then// Special check for passing parameter
10:            fParam ← map var to formal parameter at called function callee
11:            isMutate ←ReadWriteAnalysis.ISMUTATED(fParam, callee)
12:            isReturn ←ReturnAnalysis.ISRETURNED(fParam, callee)
13:            if ¬isMutate AND ¬isReturn then
14:                return true// Copy can be removed
15:            end if
16:        end if
17:    end if
18:    return false// Copy is needed in all other cases
19: end procedure
```

Table 5.1 shows the rules to remove a copy of function parameter. Whiley uses copy semantics for every array, but for an assignment $a = copy(b)$ or function call $a = func(copy(b))$ the array copy is unnecessary when:

- b is dead (not used) afterwards, or
- b is passed as read-only parameter.

The copy analyser first initialises read-write, return and live variable analysers, and then store all mutable, return and liveness sets for each function. Secondly, the copy analyser detects what copies can be eliminated using backward live variable analysis along with a decision procedure (see Algorithm 5.6). This removes copies of dead variables, which are not used afterwards, and read-only and not returned function parameters. But the copies of structure typed variables are conservatively kept avoiding memory aliases.

For each line of code in a function, our copy elimination analysis iterates through every array variable on the right-handed side, and checks if the copy can be removed and then passes the resulting flag to the code generator to produce the corresponding C code.

If the copy of a variable is removed and aliased to an existing read-write or return variable, then we will update such aliasing information to read-write and return sets to ensure the copy analyser gets updated and copy-optimised function analysis results.

5.3 Reverse Example

```
1  // Reverse an array
2  function reverse(int[] arr) -> int[]:
3    int i = |arr|
4    int[] r = [0; |arr|]
5    while i > 0 where i <= |arr| && |r| == |arr|:
6      int item = arr[|arr|-i]
7      i = i - 1
8      r[i] = item
9    return r
10 // Main entry point
11 method main(System.Console sys):
12   int[] input = [0;10] // Generated an array 'input'
13   int index = 0
14   while index < 10:
15     input[index] = 10 - index// Fill in the array (10, 9, 8, 7, ..., 2, 1)
16     index = index + 1
17   // Re−order the array
18   int[] tmp = reverse(~~copy~~(input))
19   // Check the first element of input array
20   assert input[0] == 10
21   int[] output = ~~copy~~(tmp)
22   // Check the first element of output array
23   assert output[0] == 1
24   return
```

Listing 5.1: Reverse Whiley program

Example *reverse* program (See Listing 5.1) takes an array as input and produces an array in its backward order. The main function has two copies (at line 18 and 21) to ensure the mutability of input/output arrays whereas reverse sub-function does not involve any copy. To decide the necessity of each copy, we apply copy analysis at byte-code level of Whiley program to eliminate unused copies.

Table 5.2: Live variable analysis result

Program Point	out	use	def	$in = use \cup (out - def)$
L24: return	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
L23: assert output[0] == 1	$\emptyset$	$\{output\}$	$\emptyset$	$\{output\}$
L21: int[] output = ~~copy~~(tmp)	$\{output\}$	$\{tmp\}$	$\{output\}$	$\{tmp\}$
L20: assert input[0] == 10	$\{tmp\}$	$\{input\}$	$\emptyset$	$\{input,\ tmp\}$
L18: int[] tmp = reverse(~~copy~~(input))	$\{input,\ tmp\}$	$\{input\}$	$\{tmp\}$	$\{input\}$
L16: index = index + 1	$\{input\}$	$\{index\}$	$\{index\}$	$\{input, index\}$
L15: input[index] = 10 - index	$\{input, index\}$	$\{index\}$	$\emptyset$	$\{input, index\}$
L14: while index < 10	$\{input, index\}$	$\{index\}$	$\emptyset$	$\{input, index\}$
L13: int index = 0	$\{input, index\}$	$\emptyset$	$\{index\}$	$\{input\}$
L12: int[] input= [0;100]	$\{input\}$	$\emptyset$	$\{input\}$	$\emptyset$

The copy elimination analysis first requires the backward liveness information of main function. In the live variable analysis, we enable assertion flag (`-ea`) to analyse assert code (see L20 and L23), and consider the updated array variable (see L15) as live.

Table 5.2 shows the list of live variable sets and can be used to identify whether variables are live at each program point. Note that the last return code has an empty output set and the first array generator code has an empty input set. Both are consistent to the scope of variable declaration at method *main*. We then can base on live variable analysis results to safely remove the copy of array *tmp* at L21 as *tmp* is not used afterwards.

However, to make the removal decision of another copy at L18, we need not only liveness of array *input* at method *main* but also function analysis results at function *reverse* as follows:

- Array *input* is read-only at function *reverse*.
- Array *input* is not returned by at function *reverse*.
- Array *input* is live at L19 at method *main*.

According to copy elimination rule (see Table5.1), the copy of Array *input* can be safely removed.

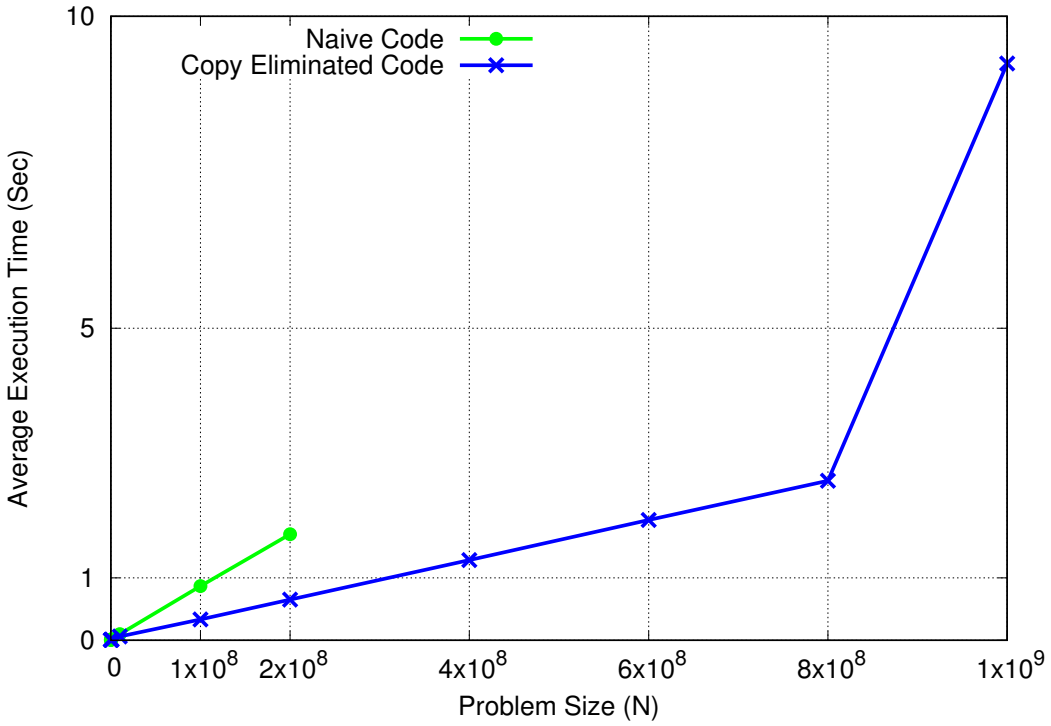


Figure 5.1: Average execution time graph of naive and copy eliminated *Reverse* program

The reverse example is translated into C code with/without copy analysis and then bench-marked on Intel i7-4770 CPU (@3.4 GHz) machine with 16 GB memory. As shown in Figure 5.1, the copy eliminated code optimised by our copy analysis remove unnecessary array copies and gain better speed-ups, without causing side effects or violating program safety. Moreover, the copy eliminated code uses less memory and increases program scalability to run for long.

Chapter 6

Memory Deallocation Analysis

The arrays or compound structures are declared as pointers in generated C code. As these data structures are dynamically allocated and explicitly de-allocated on the heap memory, any incorrect memory error leads to critical safety problems, e.g. memory leaks or double freeing.

Intuitively any previously allocated variable, which is no longer used but still bound to a memory space, needs the memory de-allocation before function exit. To determine whether the allocated variable can be safely released or not, an extra run-time *de-allocation flag* is added to each variable and its boolean value changes as the program iterates each code. At the function exit, the program checks each flag and de-allocates the corresponding variable. Note that the *array size* is another extra run-time flag, to explicitly indicate the length of an array variable and propagate the array size to a function call.

The de-allocation analyser (Weng et al., 2017) takes WyIL code as input, and adds the *pre-deallocation* and *post-deallocation* macros to change the flag value at run-time. Pre-deallocation macro targets the left-handed variable at each code to check its de-allocation flag and free the memory space. After each code, the analyser adds post-deallocation macro to bases analysis results of the code to change the flag, but still maintains the de-allocation invariant.

6.1 Deallocation Invariant

Theorem 6.1 ***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:*

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

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)`

$$\begin{aligned} &(e(a_{dealloc}) \wedge e(a) \neq NULL) \\ &\implies (\forall j : VARS \bullet (j \neq a \wedge e(j) == e(a)) \\ &\quad \implies e(j_{dealloc}) = false) \end{aligned}$$

This deallocation invariant ensures that at any program point at most one variable has the deallocation flag set to true, which allows freeing the allocated memory space. This invariant enables multiple variables to share the same allocated memory space but restricts only one variable to be responsible for de-allocating the memory structure.

The deallocation invariant is similar to the single ownership principle in Rust (Blandy, 2015): every array is bound to a single owner variable that has true flag at any given time, and when the owner is dropped, the array is deleted. But our deallocation flag is only used to indicate which variable is responsible for de-allocation purpose, and does not have control over read or write access.

6.2 Deallocation Macros

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

6.2.1 Pre-Deallocation Macro

`PRE_DEALLOC` macro empties the left-hand side variable prior to a code, so avoids any memory leak caused by the update. Any time that the value of an allocated variable is about to be overwritten, Our macro checks the flag and determines whether the variable is responsible to free that memory space, as below.

```
1 // Free variable 'a' if its deallocation flag is true
2 #define PRE_DEALLOC(a)
3 {
4   if(a_dealloc){
5     free(a);
6     a:= NULL;
7     a_dealloc:=false;
8   }
9 }
```

However, when encountering *return* code the macro is applied on all previously allocated variables (excluding the return variable), to reclaim all unused memory before the function exit.

6.2.2 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:

Array Generator An array generator a := [value;size] creates a new array of given *size* and initial *value* of each array item, and stores the new array to a variable. We define below `NEW1DARRAY_DEALLOC` macro to create a new array and check if the array is successfully allocated in memory and then populate the array by using a loop.

```
1  // Create an array of given type and size, and fill in given value
2  #define NEW1DARRAY_DEALLOC(a, value, size, type)
3  {
4    PRE_DEALLOC(a);
5    a_size := size;
6    a := (type*)malloc(a_size*sizeof(type));
7    if(a == NULL){
8      fputs("fail to allocate the memory\n", stderr);
9      exit(-2);
10   }
11   // Initialize each item value of array 'a'
12   for(size_t i:=0;i<a_size;i++){
13     a[i] := value;
14   }
15   a_dealloc := true;
16 }
```

NEW1DARRAY_DEALLOC macro includes PRE_DEALLOC macro to free the target variable before array generation, and then assigns true flag to target variable because the macro creates a fresh array address.

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  {
3    PRE_DEALLOC(a);
4    a := copy(b);
5    a_dealloc := true;
6  }
```

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  {
3    PRE_DEALLOC(a);
4    a := b;
5    a_dealloc := b_dealloc;
6    b_dealloc := false;
7  }
```

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

This macro is similar to move semantics in Rust (Blandy, 2015). Assignment in most of Rust types moves the value from one owner to another, and assigns ownership to new destination and leaves the old source unused and void. By combining single ownership rule, Rust compiler can estimate the lifetime of every variable and drop every value which does not have ownership, so that dangling pointers can never be used.

Our project also integrates similar but less restrictive move ownership to transfer the de-allocation flag from source to destination. But other aliased pointers are allowed to access the shared memory. Because the flag is transferred out during assignments, the double deallocation can be avoided.

Function Call A function call passes parameters to the called function (callee) and then returns the result back to caller site. As a call *may* or *may not* create a copy of each 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?

Table 6.1: Post-deallocation macro for function call

Function call $a := f(b)$ where a is function return and b is parameter					
f **mutates** b?		F	F	T('may-be')	T('may-be')
f **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`	Copy `CALLER_DEALLOC`	Copy `CALLER_DEALLOC`	Copy `CALLEE_DEALLOC`

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

callee. The flag value is determined by taking account of *mutable*, *return* and *liveness* analysis as shown in Table 6.1. Note 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 call.

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

`CALLER_DEALLOC` macro is applied when the parameter is or may be returned by the call and avoids being freed by callee. Due to over-approximation of return analysis, this macro would make an extra copy and lead to potential memory leaks. For example, the called 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, and 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 run-time 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);
  a_dealloc := true;// No change to 'b_dealloc'
}
```

`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 copying. 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);
  a := func(b, false); // Do not free 'b' at 'func'
  a_dealloc := true;// No change to 'b_dealloc'
}
```

`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);
  a := func(b, false);// Do not free 'b' at 'func'
  if(a != b){
    a_dealloc := true;
  }else{
    a_dealloc := b_dealloc;// Transfer 'b' flag to 'a'
    b_dealloc := false;
  }
}
```

`RESET_DEALLOC` macro is applied when the passing parameter is or may be returned by called function, so specifies the de-allocation flag at caller site.

The macro includes an aliasing check to determine flag values after the call. If the parameter is returned and thus is aliased to result, the flag is transferred out from parameter to function return. If not, a new flag is assigned to function result as the call returns a new and fresh memory space. Our macro is similar to shared or mutable *borrow reference* in Rust (Blandy, 2015). `RETAIN_DEALLOC` macro uses shared reference as the passed parameter is read-only and does not allow the called function to modify or drop its value. The mutable passed-by-reference parameter is used in `RESET_DEALLOC` macro to provide read-write access for the called function to change its value.

6.3 Informal Proofs

Our macros take variables as arguments and make changes to run-time deallocation flags and variable values. Each macro is designed to preserve a deallocation invariant before and after each execution, and ensures that only one variable is responsible for freeing one allocated memory space. To prove this, we provide the following informal proofs by using deductive reasoning.

Definition 6.1 *Our deallocation analysis supports three data types: integer (`int`), Boolean value (`bool`) and one dimensional integer array (`int[]`).*

$\mathbb{B}$ is a set of Boolean values for variables having `bool` type, i.e. $\{true, false\}$.

$\mathbb{Z}$ is a set of integers for variable having `int` type.

ADR is a set of memory addresses for variables having `int[]` type, which each points to the value of an array. $NULL$ is a special address ($NULL \in ADR$), and used to indicate an invalid address.

Let VARS be the set of variables of all supported types, including integer, Boolean and integer array.

Let ARRVARS be the set of integer array variables ($ARRVARS \subset VARS$).

Let VALUES be the value space which consists of all the sets of variable values ($VALUES = \mathbb{B} \cup \mathbb{Z} \cup ADR$).

Let `e` be a function which maps a variable to its value:

$$e : VARS \rightarrow VALUES$$

$$ARRVARS \rightarrow ADR$$

Function e bases on variable type to get the value:

- *$e(i)$ can be a value such as true or 1, if i is a Boolean or integer variable.*

$$\forall i \in VARS \bullet e(i) \in VALUES \tag{6.1}$$

- *$e(i)$ can be the address of an array, if i is an integer array typed variable.*

$$\forall i \in ARRVARS \bullet e(i) \in ADR \tag{6.2}$$

Definition 6.2 *Let $i, j \in ARRVARS$ be array variables. $i \equiv j$ means variable i and j are the same variables. If $i \equiv j$, then i and j are aliased to the same address:*

$$i \equiv j \implies e(i) = e(j) \tag{6.3}$$

However, $e(i) = e(j)$ does not guarantee $i \equiv j$.

Definition 6.3 *Let $i \in ARRVARS$ be array variable and $fresh(i)$ stand for a predicate that describes variable i and satisfies:*

$$fresh(i) : \forall j \in ARRVARS \bullet e(j) = e(i) \implies j \equiv i \tag{6.4}$$

or equivalently

$$fresh(i) : \forall j \in ARRVARS \bullet j \not\equiv i \implies e(j) \neq e(i) \tag{6.5}$$

Definition 6.4 *Let* `valid` *be a function which maps the address to true or $false$. $valid(d)$ means d is a valid address, returned from* `malloc` *function and not yet freed.*

For $x \in ARRVARS$, we have $valid(e(x))$ or $\neg valid(e(x))$. What we know about valid function are:

- *if $e(x)$ is NULL, then we have false value*

$$\neg valid(NULL) \tag{6.6}$$

- *after* `malloc` *function call, we have a fresh and valid address*

$$\{\} \quad x = malloc() \quad \{valid(e(x)) \wedge fresh(x)\} \tag{6.7}$$

- *after* `free` *function call, we have invalid address*

$$\{valid(e(x))\} \quad free(x) \quad \{\neg valid(e(x))\} \tag{6.8}$$

- *after making a copy of another array variable y, we have a fresh and valid address*

$$\{valid(e(y))\} \quad x = copy(y) \quad \{valid(e(x)) \wedge fresh(x)\} \tag{6.9}$$

Note that variable y is a valid address before the copy is made.

Assume the deallocation invariant (see Theorem 6.1) holds before a macro. After applying the macro, we still have the invariant.

Definition 6.5 *Let* $i, j \in VARS \wedge i \not\equiv j$ *and*

$$inv\_dealloc(i,j) : e(i_{dealloc}) \wedge e(j_{dealloc}) \wedge e(i) = e(j) \implies i \equiv j \tag{6.10}$$

or equivalently,

$$inv\_dealloc(i,j) : e(i_{dealloc}) \wedge e(j_{dealloc}) \wedge i \not\equiv j \implies e(i) \neq e(j) \tag{6.11}$$

stand for deallocation invariant of variable i, j*. As the invariant is symmetric, we have* $inv\_dealloc(i,j) \equiv inv\_dealloc(j,i)$*.*

Also, we include array invariant to ensure any array variable $i \in ARRVAR$ *with true flag points to a valid address:*

$$inv\_arr(i) : e(i_{dealloc}) \implies valid(e(i)) \tag{6.12}$$

So the deallocation invariant can be represented as:

$$INV : \forall i, j \in VARS \bullet inv\_dealloc(i,j) \wedge \tag{6.13a}$$

$$\forall i \in ARRVARS \bullet inv\_arr(i) \tag{6.13b}$$

6.3.1 Pre-Deallocation Macro

```
1 #define PRE_DEALLOC(a)
2 {
3   if(a_dealloc){
4     free(a); a:=NULL; a_dealloc:=false;
5   }
6 }
```

`PRE_DEALLOC` macro aims to free out the existing value of a variable and resets its flag, and leads to below proposition:

$$e(a_{dealloc}) = false \tag{6.14a}$$

$$e(a) = NULL \tag{6.14b}$$

`PRE_DEALLOC` macro is the only way of freeing a variable and avoid the double free problem in C (the same memory space is de-allocated twice).

Theorem 6.2 *If INV is true before* `PRE_DEALLOC` *macro, then INV is still true after the macro, as the below Hoare logic:*

$$\{INV \wedge (\forall i \in VARS \bullet e(i) = e_0(i)) \wedge (\forall d \in ADR \bullet valid(d) = v_0(d))\}$$

$$PRE\_DEALLOC(a)$$

$$\{INV \wedge (\forall i \in VARS \bullet i \not\equiv a \wedge i \not\equiv a_{dealloc} \implies e(i) = e_0(i))$$

$$\wedge (\forall d \in ADR \bullet d \neq e_0(a) \implies valid(d) = v_0(d)) \wedge \neg e(a_{dealloc})\}$$

The precondition stores variable address and validity in the pre-states with $e_0(i)$ and v_0 respectively. And the post-condition ensures all array variables, except for a, remain the same address and validity.

```
1  {INV ∧ (∀i ∈ VARS • e(i) = e0(i)) ∧ (∀d ∈ ADR • valid(d) = v0(d))}
2  if(a_dealloc){
3
4     {INV ∧ (∀i ∈ VARS • e(i) = e0(i)) ∧ (∀d ∈ ADR • valid(d) = v0(d))
5              ∧e(a_deallc) ∧ valid(e(a))}①
6     free(a);
7     a:=NULL;
8     a_dealloc:=false;
9     {(∀i ∈ VARS • i ≢ a ∧ i ≢ a_dealloc ⟹ e(i) = e0(i))②a
10         ∧(∀d ∈ ADR • d ≠ e0(a) ⟹ valid(d) = v0(d))②b∧
11         ¬e(a_dealloc) ∧ INV②c}
12
13 }
14
15 {INV ∧ (∀i ∈ VARS • i ≢ a ∧ i ≢ a_dealloc ⟹ e(i) = e0(i))
16     ∧(∀d ∈ ADR • d ≠ e0(a) ⟹ valid(d) = v0(d) ∧ ¬e(a_dealloc))③
```

Listing 6.1: Tableau of `PRE_DEALLOC(a)` macro

Reasoning about ① Show that $valid(e(a))$ holds true before `free` code because INV is true, which implies $inv\_arr(a)$ is true.

$$inv\_arr(a) = e(a_{dealloc}) \implies valid(e(a))$$

Also $e(a_{dealloc})$ holds true (in line 3), so we have $valid(e(a))$ in ①

Assumption ②a and ②b Assumes $e(i) = e_0(i)$ and $valid(d) = v_0(d)$ is true. Since line 5 to 7 changes variable a and a's flag, the other variables remain unchanged.

Reasoning about (2c) Show that INV holds at the end of IF branch.

$$INV : \forall i, j \in VARS \bullet inv\_dealloc(i, j)Ⓐ \wedge \forall i \in ARRVARS \bullet inv\_arr(i)Ⓑ$$

Proof. [Reasoning Deallocation Invariant Ⓐ]
Let $i, j \in VARS$ and $i \neq j$ be the witness variables. Assume that $\forall i, j \in VARS \bullet inv\_dealloc(i, j))$ holds true before `PRE_DEALLOC` macro. Consider the following three cases:

- Case 1: Only i include a

  Given $i \equiv a \wedge j \not\equiv a$, we can replace variable i with a and write the invariant as below:

  $$\begin{aligned} inv\_dealloc(a, j) \iff & (e(a_{dealloc}) \wedge e(j_{dealloc}) \\ & \wedge a \not\equiv j) \implies e(a) \neq e(j) \end{aligned} \tag{6.15}$$

  From Proposition 6.14a, we evaluate the predicate of $inv\_dealloc(a, j)$ to be $false$ and thus produces the $true$ value of $inv\_dealloc(a, j)$.

- Case 2: Only j includes a

  This case is the same as Case 1 that a is one of i or j ($inv\_dealloc(i, a) \iff inv\_dealloc(a, i)$). As $inv\_dealloc(a, j)$ is proven to be true in Case 1, we can conclude $inv\_dealloc(i, a)$ also holds true.

- Case 3: i, j NOT include a

  In this case $i \not\equiv a \wedge j \not\equiv a$, our macro does not change any variable, so the invariant $inv\_dealloc(i, j)$ still holds true.

□

Proof. [Reasoning Array Invariant Ⓑ]
We must show $\forall i \in ARRVARS \bullet inv\_arr(i) = e(i_{dealloc}) \implies valid(e(i))$ holds true in the post-state.

Let $i \in ARRVARS$ such that $e(i_{dealloc})$ is true in the post-state and we have to show $valid(e(i))$.

We have $\neg e(a_{dealloc})$ and $e(i_{dealloc})$. So $a \not\equiv i$.

For $a \not\equiv i$, we know variable i's flag is unchanged $e(i_{dealloc}) = e_0(i_{dealloc})$, which we assume it is true.

Also, variable a's flag is true $e_0(a_{dealloc})$ from entry condition of IF branch.

So $e_0(a_{dealloc}) \wedge e_0(i_{dealloc}) \wedge a \not\equiv i$.

Because INV holds before the macro (using e_0 instead of e), we have

$$e_0(a_{dealloc}) \wedge e_0(i_{dealloc}) \wedge a \not\equiv i \implies e_0(a) \neq e_0(i)$$

Therefore, $e_0(a) \neq e_0(i)$

From (2a), we also know that the validity of all addresses, except for $e_0(a)$, is unchanged.

$$e_0(i) \neq e_0(a) \implies valid(e_0(i)) = v_0(e(i)) \tag{6.16}$$

Also from INV in the pre-state

$$e_0(i_{dealloc}) \implies v_0(e(i)) \tag{6.17}$$

And because $i \not\equiv a$ and from (2b) we know that $e(i)$ is unchanged.

$$e(i) = e_0(i) \implies valid(e(i)) = valid(e_0(i)) = v_0(e(i)) = true \tag{6.18}$$

□

Reasoning about ③ Show INV holds true in the post condition because INV is true in IF branch (see (2c)). At ③, we use the condition in IF branch and skip the one in ELSE branch, which has not defined yet. So we have

$$\{INV \wedge (\forall i \in VARS \bullet i \not\equiv a \wedge i \not\equiv a_{dealloc} \implies e(i) = e_0(i))$$
$$\wedge (\forall d \in ADR \bullet d \neq e_0(a) \implies valid(d) = v_0(d))$$

Also, we have $\neg e(a_{dealloc})$ in the post condition because the conditional branch gives $a_{dealloc}$ false value.

6.3.2 Array Generator

An array generator arraygenerator(a = [value;size]) creates an array a of given *size* and *value*.

```
1  #define NEW1DARRAY_DEALLOC(a, value, size, type)
2  {
3    PRE_DEALLOC(a);
4    a_size = size;
5    a = (type*)malloc(a_size*sizeof(type));
6    if(a == NULL){
7      fputs("fail to allocate memory\n", stderr);
8      exit(-2);
9    }
10   // Initialize each item value of array 'a'
11   for(size_t i=0;i<a_size;i++){
12     a[i] = value;
13   }
14   a_dealloc := true;
15 }
```

`NEW1DARRAY_DEALLOC(a, value, size)` macro:

- uses pre-deallocation macro to empty variable a;
- use `NEW_1DARRAY` macro to create a fresh array of given *size* and initialise the value of each array item;
- assigns value true to *a_dealloc* flag to indicate variable a is responsible for the de-allocation of this newly created array.

Assumption 1 *For an array generator $a := [value; size]$, we include a pre-condition that INV is true before `NEW1DARRAY_DEALLOC` macro.*

With this precondition, we can ensure there is a single deallocation owner each array variable. Thus, using `PRE_DEALLOC(a)` macro to release the memory of array variable a will not cause double freeing problem.

Theorem 6.3 *`NEW1DARRAY_DEALLOC(a, value, size)` macro creates a new array of given size and value, and then assigns to a.*

If INV holds before `NEW1DARRAY_DEALLOC(a, value, size)` macro, then

INV still holds true after the macro, as below Hoare Logic:

$$\{INV \wedge (\forall i \in VARS \bullet e(i) = e_0(i))$$
$$\wedge (\forall d \in ADR \bullet valid(d) = v_0(d))\}$$
`NEW1DARRAY_DEALLOC(a, value, size)`
$$\{INV \wedge (\forall i \in VARS \bullet i \not\equiv a \wedge i \not\equiv a_{dealloc} \implies e(i) = e_0(i))$$
$$\wedge (\forall d \in ADR \bullet d \neq e_0(a) \wedge d \neq e(a) \implies valid(d) = v_0(d))$$
$$\wedge valid(e(a))\}$$

As *INV* is preserved before and after pre-deallocation macro, we only need to prove our invariant still holds after `NEW1DARRAY_DEALLOC` macro, as below:

```
1  {INV ∧ (∀i ∈ VARS • e(i) = e0(i)) ∧ (∀d ∈ ADR • valid(d) = v0(d))}①
2  PRE_DEALLOC(a);
3  {INV ∧ (∀i ∈ VARS • i ≢ a ∧ i ≢ a_dealloc ⟹ e(i) = e0(i))
4       ∧(∀d ∈ ADR • d ≠ e0(a) ⟹ valid(d) = v0(d))}②
5
6  {INV ∧ (∀i ∈ VARS • e(i) = e1(i))
7       ∧(∀d ∈ ADR • valid(d) = v1(d))③
8
9  NEW_1DARRAY(a, value, size, int64_t);
10 a_dealloc:=true;
11
12 {(∀i ∈ VARS • i ≢ a ∧ i ≢ a_dealloc ⟹ e(i) = e1(i))④a∧
13      (∀d ∈ ADR • d ≠ e(a) ⟹ valid(d) = v1(d))④b∧
14      fresh(a) ∧ valid(e(a)) ∧ e(a_dealloc)④c}
15
16 {INV⑤a ∧ (∀i ∈ VARS • i ≢ a ∧ i ≢ a_dealloc ⟹ e(i) = e0(i))⑤b∧
17      (∀d ∈ ADR • d ≠ e0(a) ∧ d ≠ e(a) ⟹ valid(d) = v0(d))⑤c∧
18      valid(e(a)) ∧ e(a_dealloc)}
```

Listing 6.2: Tableau of `NEW1DARRAY_DEALLOC(a, value, size)` macro

Assumption ① *INV* $e(i) = e_0(i)$ and $valid(d) = v_0(d)$ are assumed to be true in the entry condition of the macro.

Reasoning about ② Show *INV* $e(i) = e_0(i)$ and $valid(d) = v_0(d)$ hold true in the post condition of pre-deallocation macro (refer to Theorem 6.2).

Assumption ③ Define $e_1(i)$ and $v_1(d)$ to store the addresses and validity of variables respectively after `PRE_DEALLOC` macro. By doing so, we can focus on the pre-and post-conditions of line 8 and 9.

Reasoning about (4a) and (4b) Show $e(i) = e_1(i)$ and $valid(d) = v_1(i)$ is true. Because our macro creates a fresh address for variable a, $e(a)$ and $a_{dealloc}$ are changed by line 8 and 9. For all other variables, $e(i)$ remains the same as $e_1(i)$ and the validity is unchanged as $v_1(i)$ in line 7 for all addresses, except for $e(a)$.

Assumption (4c) Show $fresh(a)$ $valid(e(a))$ and $e(a_{dealloc})$ are true in the post-condition. $fresh(a) \wedge valid(e(a))$ is included into the post-state because of malloc function calls used in $NEW\_1DARRAY$ macro (see Definition 6.7). $e(a_{dealloc})$ is true from line 9.

Reasoning about (5a) Show that INV holds at the end of macro.

$$INV : \forall i, j \in VARS \bullet inv\_dealloc(i, j)\text{Ⓐ} \wedge \forall i \in ARRVARS \bullet inv\_arr(i)\text{Ⓑ}$$

Proof. [Reasoning Deallocation Invariant (Ⓐ)]
Let $i, j \in ARRVARS$ such that $e(i_{dealloc})$ and $e(j_{dealloc})$ and $i \not\equiv j$. We must show that $inv\_dealloc(i, j)$ holds true after `NEW1DARRAY_DEALLOC(a, value, size)` macro.

Consider the following three cases:

- Case 1: $i \equiv a$

  Given $i \equiv a \wedge j \not\equiv a$, we can replace variable i with a and write the invariant as below:

  $$inv\_dealloc(a, j) \iff (e(a_{dealloc}) \wedge e(j_{dealloc}) \wedge a \not\equiv j) \implies e(a) \neq e(j)$$

  Because of $fresh(a)$ at (4c), we have $j \not\equiv a \implies e(j) \neq e(a)$ hold true. Thus, we can conclude $inv\_dealloc(a, j)$ is true in the post state.

- Case 2: $j \equiv a$

  This case is the same as Case 1 that a is one of i or j

  $$inv\_dealloc(i, a) \iff inv\_dealloc(a, i)$$

Since $inv\_dealloc(a, j)$ is proven to be true in Case 1, we can conclude $inv\_dealloc(i, a)$ also holds true.

- Case 3: i, j NOT include a

  Because $i \not\equiv a \wedge j \not\equiv a$, our macro changes a and $a\_dealloc$ but keeps variable i or j unchanged, and because INV holds at ③, therefore $inv\_dealloc(i, j)$ still holds true.

□

Proof. [Reasoning Array Invariant Ⓑ]

We must show $\forall i \in ARRVARS \bullet e(i_{dealloc}) \implies valid(e(i))$ holds true in the post-state.

Let $i \in ARRVARS$ such that $e(i_{dealloc})$ is true in the post state and we have to show $valid(e(i))$.

- Case 1: $i \not\equiv a$

  We know $e(i) \neq e(a)$ because of $fresh(a)$.

  From $inv\_arr(i)$ at ③, we have

  $$\forall i \in ARRVARS \bullet e_1(i_{dealloc}) \implies v_1(e_1(i))$$

  From ④ₐ, we have $e(i) = e_1(i)$ because $i \not\equiv a \wedge i \not\equiv a_{dealloc}$.

  From ④ᵦ, we have $valid(e(i)) = v_1(e(i))$ because $e(i) \neq e(a)$

  Therefore, the validity must remain unchanged in the post-state.

  $$valid(e(i)) = v_1(e(i)) = v_1(e_1(i))$$

- Case 2: $i \equiv a$

  $inv\_arr(a) : e(a_{dealloc}) \implies valid(e(a))$ holds true because we have $valid(e(a))$ in ④꜀ from the post condition of malloc function call (see Definition 6.7)

□

Reasoning about (5b) Show $\forall i \in VARS \bullet i \not\equiv a \wedge i \not\equiv a_{dealloc} \implies e(i) = e_0(i)$ is true in the post condition.

Proof. Let $i \in VARS$ be a variable such that $i \not\equiv a$ and $i \not\equiv a_{dealloc}$. We must show $e(i) = e_0(i)$.

From (4a) because $i \not\equiv a \wedge i \not\equiv a_{dealloc}$, we have $e(i) = e_1(i)$.

From (2) and (3) because $i \not\equiv a \wedge i \not\equiv a_{dealloc}$, we have $e_0(i) = e_1(i)$.

Therefore, by combining the above conditions $i \not\equiv a \wedge i \not\equiv a_{dealloc}$, we conclude $e(i) = e_0(i)$.

□

Reasoning about (5c) Show $\forall d \in ADR \bullet d \neq e_0(a) \wedge d \neq e(a) \implies valid(d) = v_0(d)$ is true in the post condition.

Proof. Let $d \in ADR$ be an address such that $d \neq e_0(a)$ and $d \neq e(a)$. We must show $valid(d) = v_0(d)$.

From (4b) because $d \neq e(a)$, we have $valid(d) = v_1(d)$.

From (2) and (3) because $d \neq e_0(a)$, we have $v_0(d) = v_1(d)$.

Therefore, by combining the above conditions $(d \neq e_0(a) \wedge d \neq e(a))$, we conclude $valid(d) = v_1(d) = v_0(d)$.

□

6.3.3 Assignment

For an assignment a := b, our deallocation analyser uses the live variable analysis to decide whether to remove the copy at right-handed side.

6.3.3.1 ADD_DEALLOC Macro

```
1 #define ADD_DEALLOC(a, b)
2 {
3   PRE_DEALLOC(a);
4   a := copy(b);
5   a_dealloc := true; // Add the Deallocation to 'a'
6 }
```

`ADD_DEALLOC(a, b)` macro:

- uses pre-deallocation macro to empty variable a and reset its flag value;
- creates a fresh copy of variable b, whose memory space is not aliased with any existing variable;
- assigns the copied b to variable a and add the flag to a

Assumption 2 *For an assignment $a := copy(b)$, we include a precondition*

$$e(a) \neq e(b)$$

to ensure when $a_{dealloc}$ and $b_{dealloc}$ are both true, variable a and b are not aliased to the same memory space before the function call. Also, we need an extra precondition

$$valid(e(b)) = true$$

to ensure the memory address pointed by variable b is valid and safe to perform the operation. With above preconditions, we can ensure $a_{dealloc}$ must be false when variable a and b are aliased and also $b_{dealloc}$ is true. Therefore, ***`PRE_DEALLOC(a)`*** *will not free the memory at $e(a) = e(b)$, making $e(b)$ an invalid address and avoid segmentation fault when trying to copy from $e(b)$.*

Theorem 6.4 ***`ADD_DEALLOC(a, b)`*** *macro makes a copy of variable b and assigns it to a. If INV and $e(a) \neq e(b)$ and $valid(e(b))$ hold before* ***`ADD_DEALLOC(a, b)`*** *macro, then INV still holds true after the macro, as below Hoare Logic:*

$$\{INV \wedge (\forall i \in VARS \bullet e(i) = e_0(i)) \wedge (\forall d \in ADR \bullet valid(d) = v_0(d))$$
$$\wedge\, e(a) \neq e(b) \wedge valid(e(b))\}$$

`ADD_DEALLOC(a, b)`

$$\{INV \wedge (\forall i \in VARS \bullet i \not\equiv a \wedge i \not\equiv a_{dealloc} \implies e(i) = e_0(i))$$
$$\wedge\, (\forall d \in ADR \bullet d \neq e_0(a) \wedge d \neq e(a) \implies valid(d) = v_0(d))$$
$$\wedge\, valid(e(a)) \wedge valid(e(b))\}$$

1 $\{INV \wedge (\forall i \in VARS \bullet e(i) = e_0(i)) \wedge (\forall d \in ADR \bullet valid(d) = v_0(d)) \wedge$
2 $\quad e(a) \neq e(b) \wedge valid(e(b))\}$①
3 *PRE_DEALLOC(a);*
4 $\{INV \wedge (\forall i \in VARS \bullet i \not\equiv a \wedge i \not\equiv a_{dealloc} \implies e(i) = e_0(i))$
5 $\quad \wedge(\forall d \in ADR \bullet d \neq e_0(a) \implies valid(d) = v_0(d)) \wedge$
6 $\quad e_0(a) \neq e(b) \wedge valid(e(b))\}$②
7
8 $\{INV \wedge valid(e(b)) \wedge (\forall i \in VARS \bullet e(i) = e_1(i))$
9 $\quad \wedge(\forall d \in ADR \bullet valid(d) = v_1(d))$③
10 *a:=copy(b);*
11 *a_dealloc:=***true***;*
12 $\{fresh(a) \wedge (\forall i \in VARS \bullet i \not\equiv a \wedge i \not\equiv a_{dealloc} \implies e(i) = e_1(i))$④a$\wedge$
13 $\quad (\forall d \in ADR \bullet d \neq e(a) \implies valid(d) = v_1(d))$④b$\wedge$
14 $\quad valid(e(a)) \wedge valid(e(b)) \wedge e(a_{dealloc})$④c$\}$
15
16 $\{INV$⑤a$\wedge (\forall i \in VARS \bullet i \not\equiv a \wedge i \not\equiv a_{dealloc} \implies e(i) = e_0(i))$⑤b$\wedge$
17 $\quad (\forall d \in ADR \bullet d \neq e_0(a) \wedge d \neq e(a) \implies valid(d) = v_0(d))$⑤c$\wedge$
18 $\quad valid(e(b)) \wedge valid(e(a)) \wedge e(a_{dealloc})\}$

Listing 6.3: Tableau of `ADD_DEALLOC(a, b)` macro

The previous section shows our invariant is preserved before and after pre-deallocation macro. So we are only required to prove our invariant still holds after the last two code statements, as below:

Assumption ① $e(a) \neq e(b)$ and $valid(e(b))$ are assumed to be true in the entry condition of the macro.

Reasoning about ② Show $e(b) = e_0(a)$ and $valid(e(b))$ are both true in the post condition of pre-deallocation macro (refer to Theorem 6.2).

Because $e(b) = e_0(b) \neq e_0(a)$ and only validity of $e_0(a)$ is changed by pre-deallocation macro, $valid(e(b))$ is true in the post-state.

Assumption ③ Define $e_1(i)$ and $v_1(d)$ to store the addresses and validity of variables respectively after `PRE_DEALLOC` macro. In doing so, we can focus on the precondition and post condition of line 10 and 11.

Reasoning about ④a and ④b Show $e(i) = e_1(i)$ and $valid(d) = v_1(i)$ is true. Since only $e(a)$ and $a_{dealloc}$ is changed by line 10 and 11, $e(i)$ remains the same as $e_1(i)$ for all other variables, and the validity is unchanged for all addresses except for $e(a)$ as $v_1(i)$ in line 8.

Assumption ④c Show $valid(e(a))$, $valid(e(b))$, $fresh(a)$ and $e(a_{dealloc})$ are true in the post-condition.

$valid(e(a)) \wedge fresh(a)$ is included into the post-state from 6.9.

$valide(e(b))$ remains true in the post-state because our macro in (line 10 and 11) does not de-allocate anything.

$e(a_{dealloc})$ is true from line 11.

Reasoning about ⑤a Show that INV holds at the end of macro.

$$INV : \forall i, j \in VARS \bullet inv\_dealloc(i, j)Ⓐ \wedge \forall i \in ARRVARS \bullet inv\_arr(i)Ⓑ$$

Proof. [Reasoning Deallocation Invariant (Ⓐ)]
Let $i, j \in VARS$ be the witness variables. We must show that $\forall i, j \bullet inv\_dealloc(i, j)$ holds true after `ADD_DEALLOC(a, b)` macro. Consider the following four cases:

- Case 1: i, j includes both a and b

  Given $i \equiv a \wedge j \equiv b$ (or equivalently $j \equiv a \wedge i \equiv b$), the invariant can be rewritten as:

$$\begin{aligned} inv\_dealloc(a, b) :& (e(a_{dealloc}) \wedge e(b_{dealloc}) \wedge \\ & a \not\equiv b) \implies e(a) \neq e(b) \end{aligned} \tag{6.19}$$

  From precondition $e_0(a) \neq e_0(b)$, which implies $a \not\equiv b$, and $fresh(a)$, we conclude that $a \not\equiv b \implies e(a) \neq e(b)$. That implies that $inv\_dealloc(a, b)$ still is true in the post-state.

- Case 2: i, j includes a but NOT b

  Given $i \equiv a \wedge j \not\equiv b$ (or equivalently $j \equiv a \wedge i \not\equiv b$), the invariant can be rewritten as:

$$\begin{aligned} inv\_dealloc(a, j) :& e(a_{dealloc}) \wedge e(j_{dealloc}) \wedge \\ & a \not\equiv j \implies e(a) \neq e(j) \end{aligned} \tag{6.20}$$

  Assume that all the preconditions in (6.20) are true, including $a \not\equiv j$. With true value of $fresh(a) : j \not\equiv a \implies e(j) \neq (a)$, we have $e(j) \neq e(a)$ in the post-state and conclude $inv\_dealloc(a, j)$ is true after the macro.

- Case 3: i, j includes b but NOT a

  Given $i \equiv b \wedge j \not\equiv a$ (or equivalently $j \equiv b \wedge i \not\equiv a$), the invariant can be rewritten as:

$$\begin{aligned} inv\_dealloc(b, j) :& e(b_{dealloc}) \wedge e(j_{dealloc}) \wedge \\ & b \not\equiv j \implies e(b) \neq e(j) \end{aligned} \tag{6.21}$$

  Because $j \not\equiv a$ and only variable a and $a_{dealloc}$ are changed so variable j and $j_{dealloc}$ stay unchanged in post-state.

  Since $inv\_dealloc(b, j)$ was true in the pre-state, we have $inv\_dealloc(b, j)$ hold true in the post-state.

- Case 4: i, j are both different from a, b

  The macro does not change any value of variable i and j, so the invariant $inv\_dealloc(i, j)$ still holds.

□

Proof. [Reasoning Array Invariant Ⓑ]

We must show $\forall i \in ARRVARS \bullet e(i_{dealloc}) \implies valid(e(i))$ holds true in the post-state.

Let $i \in ARRVARS$ such that $e(i_{dealloc})$ is true in the post-state and we have to show $valid(e(i))$.

- Case 1: $i \not\equiv a$

  We know $e(i) \neq e(a)$ because of $fresh(a)$.

  From $inv\_arr(i)$ in line 8, we have

$$\forall i \in ARRVARS \bullet e_1(i_{dealloc}) \implies v_1(e_1(i))$$

  From ④a, we have $e(i) = e_1(i)$ because $i \not\equiv a \wedge i \not\equiv a_{dealloc}$.

  From ④b, we have $valid(e(i)) = v_1(e(i))$ because $e(i) \neq e(a)$

  Therefore, the validity must remain unchanged in the post-state.

$$valid(e(i)) = v_1(e(i)) = v_1(e_1(i))$$

- Case 2: $i \equiv a$

  $inv\_arr(a) : e(a_{dealloc}) \implies valid(e(a))$ holds true because we have $valid(e(a))$ in ④c, which comes from post condition of *copy*.

□

Reasoning about ⑤b Show $\forall i \in VARS \bullet i \not\equiv a \wedge i \not\equiv a_{dealloc} \implies e(i) = e_0(i)$ is true in the post condition.

Proof. Let $i \in VARS$ be a variable such that $i \not\equiv a$ and $i \not\equiv a_{dealloc}$. We must show $e(i) = e_0(i)$.

From ④a because $i \not\equiv a \wedge i \not\equiv a_{dealloc}$, we have $e(i) = e_1(i)$.

From ② and ③ because $i \not\equiv a \wedge i \not\equiv a_{dealloc}$, we have $e_0(i) = e_1(i)$.

Therefore, by combining the above conditions $i \not\equiv a \wedge i \not\equiv a_{dealloc}$, we conclude $e(i) = e_0(i)$.

□

Reasoning about ⑤c Show $\forall d \in ADR \bullet d \neq e_0(a) \wedge d \neq e(a) \implies valid(d) = v_0(d)$ is true in the post condition.

Proof. Let $d \in ADR$ be an address such that $d \neq e_0(a)$ and $d \neq e(a)$. We must show $valid(d) = v_0(d)$.

From ④b because $d \neq e(a)$, we have $valid(d) = v_1(d)$.

From ② and ③ because $d \neq e_0(a)$, we have $v_0(d) = v_1(d)$.

Therefore, by combining the above conditions $(d \neq e_0(a) \wedge d \neq e(a))$, we conclude $valid(d) = v_1(d) = v_0(d)$.

□

6.3.3.2 TRANSFER_DEALLOC Macro

```
1 #define TRANSFER_DEALLOC(a, b)
2 {
3   PRE_DEALLOC(a);
4   a := b;
5   a_dealloc := b_dealloc;
6   b_dealloc := false;
7 }
```

TRANSFER_DEALLOC(a, b) macro:

- uses pre-deallocation macro to empty variable a and reset its flag value;
- aliases variable a and b, so they both point to the same memory space;
- assigns variable b's flag value to a and then reset b's flag.

Assumption 3 *For an assignment $a := b$, we include a precondition*

$$e(a) \neq e(b)$$

to ensure both $a_{dealloc}$ and $b_{dealloc}$ can not be true when variable a and b are aliased to the same memory space before the function call.

Also, we need an extra precondition

$$valid(e(b)) = true$$

to ensure the memory address pointed by variable b is valid and safe to perform the operation.

These preconditions ensure the aliased variables a and b after the macro do not point to an invalid address, and cause null-pointer de-reference exceptions when accessing the value of variable a or b.

Theorem 6.5 *Let $e_0(b)$ be the value of variable b and $e_0(b_{dealloc})$ be the flag value of variable b in the pre-state of `TRANSFER_DEALLOC(a, b)` macro.*

If INV holds before `TRANSFER_DEALLOC(a, b)` macro, then INV still holds

true after the macro, as below Hoare Logic:

$\{INV \wedge (\forall i \in VARS \bullet e(i) = e_0(i))$

$\quad \wedge (\forall d \in ADR \bullet valid(d) = v_0(d))$

$\quad \wedge e(a) \neq e(b) \wedge valid(e(b))\}$

`TRANSFER_DEALLOC(a, b)`

$\{INV \wedge (\forall i \in VARS \bullet i \not\equiv a \wedge i \not\equiv a_{dealloc} \wedge i \not\equiv b_{dealloc} \implies e(i) = e_0(i))$

$\quad \wedge (\forall d \in ADR \bullet d \neq e_0(a) \implies valid(d) = v_0(d))$

$\quad \wedge e(a) = e(b) \wedge valid(e(a)) \wedge e(a_{dealloc}) = e_0(b_{dealloc})\}$

---

1 $\{INV \wedge (\forall i \in VARS \bullet e(i) = e_0(i)) \wedge (\forall d \in ADR \bullet valid(d) = v_0(d)) \wedge$
2 $\quad e(a) \neq e(b) \wedge valid(e(b))\}$①
3 `PRE_DEALLOC(a);`
4 $\{INV \wedge (\forall i \in VARS \bullet i \not\equiv a \wedge i \not\equiv a_{dealloc} \implies e(i) = e_0(i)) \wedge$
5 $\quad (\forall d \in ADR \bullet d \neq e_0(a) \implies valid(d) = v_0(d)) \wedge$
6 $\quad e_0(a) \neq e(b) \wedge valid(e(b))\}$②
7 $\{INV \wedge valid(e(b)) \wedge (\forall i \in VARS \bullet e(i) = e_1(i)) \wedge$
8 $\quad (\forall d \in ADR \bullet valid(d) = v_1(d))\}$③
9
10 `a:=b;`
11 `a_dealloc:=b_dealloc;`
12 `b_dealloc:=false;`
13
14 $\{(\forall i \in VARS \bullet i \not\equiv a \wedge i \not\equiv a_{dealloc} \wedge i \not\equiv b_{dealloc} \implies e(i) = e_1(i))$④a$\wedge$
15 $\quad (\forall d \in ADR \bullet valid(d) = v_1(d)$④b$) \wedge$
16 $\quad e(a) = e(b) = e_1(b) \wedge valid(e(a)) \wedge e(a_{dealloc}) = e_1(b_{dealloc}) \wedge \neg e(b_{dealloc})$④c$\}$
17
18 $\{INV$⑤a$\wedge (\forall i \in VARS \bullet i \not\equiv a \wedge i \not\equiv a_{dealloc} \wedge i \not\equiv b_{dealloc} \implies e(i) = e_0(i))$⑤b
19 $\quad \wedge (\forall d \in ADR \bullet d \neq e_0(a) \implies valid(d) = v_0(d))$⑤c$\wedge$
20 $\quad e(a) = e(b) \wedge valid(e(a)) \wedge e(a_{dealloc}) = e_0(b_{dealloc})\}$

Listing 6.4: Tableau of `TRANSFER_DEALLOC(a, b)` macro

Assumption ① Assume $e(a) \neq e(b)$ and $valid(e(b))$ are true in the entry condition of the macro as we justify the precondition at start of an assignment (see Definition 2).

Reasoning about ② Show that $e_0(a) \neq e(b)$ and $valid(e(b))$ are true in post condition of the pre-deallocation macro (refer to Theorem 6.2).

Assumption ③ Define $e_1(i)$ and $v_1(d)$ to store the addresses and validity of variables respectively after `PRE_DEALLOC` macro. By doing so we can focus on the pre- and post conditions between line 10 and 12.

Reasoning about ④ Show $e(a) = e(b) = e_0(b)$ and $valid(e(a))$ are true in the post-condition. Because transfer macro does not change the validity of any variable but aliases a and b, we have $e(a) = e(b)$. Therefore, we conclude $valid(e(a)) = valid(e(b)) = valid(e_1(b)) = v_1(e_1(b)) = true$.

Reasoning about ⑤a Show that INV holds at the end of macro.

$$INV : \forall i, j \in VARS \bullet inv\_dealloc(i, j)Ⓐ \wedge \forall i \in ARRVARS \bullet inv\_arr(i)Ⓑ$$

Proof. [Reasoning Deallocation Invariant Ⓐ]
Let $i, j \in VARS$ be the witness variables. We must show $\forall i, j \bullet inv\_dealloc(i, j)$ holds true after `TRANSFER_DEALLOC(a, b)` macro.

As $inv\_dealloc(i, j)$ is symmetric, we can swap variable i and j without breaking the invariant, so $inv\_dealloc(i, j) \iff inv\_dealloc(j, i)$ and the reasoning just needs to consider three cases:

- Case 1: i, j includes b

  Given $j \equiv b$ (or equivalently $i \equiv b$), the invariant can be rewritten as:

  $$\begin{aligned} inv\_dealloc(i, b) :&(e(i_{dealloc}) \wedge e(b_{dealloc}) \wedge \\ &i \not\equiv b) \implies e(i) \neq e(b) \end{aligned} \tag{6.22}$$

  Since $e(b_{dealloc})$ is false in post-state, $inv\_dealloc(i, b)$ holds true after the macro.

- Case 2: i, j includes a but NOT b

  Given $i \equiv a \wedge j \not\equiv b$ (or equivalently $j \equiv a \wedge i \not\equiv b$), the invariant can be rewritten as:

  $$\begin{aligned} inv\_dealloc(a, j) :&(e(a_{dealloc}) \wedge e(j_{dealloc}) \wedge \\ &j \not\equiv a) \implies e(a) \neq e(j) \end{aligned} \tag{6.23}$$

  Because of $e(a_{dealloc}) = e_1(b_{dealloc})$ from ④, we can rewrite the invariant:

  $$\begin{aligned} inv\_dealloc(b, j) :&(e_1(b_{dealloc}) \wedge e(j_{dealloc}) \\ &\wedge j \not\equiv a) \implies e(a) \neq e(j) \end{aligned} \tag{6.24}$$

  where $j \not\equiv b$ by the assumption of this case

Assume the preconditions of above implication (6.24) hold in the post-state. We need to show $e(a) \neq e(j)$:

- From $e_1(b_{dealloc})$ we conclude $e(b_{dealloc})$ was true in line 8.
- From $e(j_{dealloc})$ we conclude $e(j_{dealloc})$ was true in line 8, because $j \not\equiv a, b$ and the macro only changes a and b.

Because $j \not\equiv b$ and the $inv\_dealloc(b, j)$ was true in line 8, we get $e_1(b) \neq e_1(j)$. Finally, by using $e(a) = e_1(b)$ from (4c) and $e(j) = e_1(j)$ from (4a), because $j \not\equiv b$, we have $e(a) \neq e(j)$ in the post-state. So $inv\_dealloc(b, j)$ is true after the macro.

- Case 3: i, j are both different from a, b

  This macro does not change any variable, except for a and b, so the invariant still holds.

□

Proof. [Reasoning Array Invariant (B)]

We must show $inv\_arr(i) : \forall i \in ARRVARS \bullet e(i_{dealloc}) \implies valid(e(i))$ holds true in the post-state.

Let $i \in ARRVARS$ such that $e(i_{dealloc})$ is true in the post-state and we have to show $valid(e(i))$.

- Case 1: $i \not\equiv a$

  Because $e(b_{dealloc}) = false$, we know $i \not\equiv b$.

  From (4a) because $e(i_{dealloc}) = e_1(i_{dealloc})$ we get $e_1(i_{dealloc}) = true$.

  Because INV holds true at line 8

  $$inv\_arr(i) : e_1(i\_dealloc) \implies v_1(e_1(i))$$

  We have $v_1(e_1(i)) = true$.

  Therefore, since $e(i) = e_1(i)$ from (4b), we conclude $v_1(e_1(i)) = valid(e_1(i)) = valid(e(i))$ by (4b).

- Case 2: $i \equiv a$

  We have $e(i_{dealloc}) = e(a_{dealloc}) = e_1(b_{dealloc})$ ④c.

  Because INV is true at line 9,

$$inv\_arr(b) : e_1(b_{dealloc}) \implies v_1(e_1(b))$$

  So $v_1(e_1(b)) = true$

  Also, $e(a) = e_1(b)$ from ④c, so $v_1(e(a)) = v_1(e_1(b))$. And by ④b, we have

$$valid(e(i)) = valid(e(a)) = v_1(e(a)) = v_1(e_1(b)) = true$$

□

Reasoning about ⑤b Show $\forall i \in VARS \bullet i \not\equiv a \wedge i \not\equiv a_{dealloc} \wedge b_{dealloc} \implies e(i) = e_0(i)$ is true in the post condition.

Proof. Let $i \in VARS$ be a variable such that $i \not\equiv a$ and $i \not\equiv a_{dealloc}$. We must show $e(i) = e_0(i)$.

From ④b because $i \not\equiv a \wedge i \not\equiv a_{dealloc} \wedge b_{dealloc}$, we have $e(i) = e_1(i)$.

From ② and ③ because $i \not\equiv a \wedge i \not\equiv a_{dealloc}$, we have $e_0(i) = e_1(i)$.

Therefore, by combining the above conditions $i \not\equiv a \wedge i \not\equiv a_{dealloc} \wedge b_{dealloc}$, we conclude $e(i) = e_0(i)$.

□

Reasoning about ⑤c Show $\forall d \in ADR \bullet d \neq e_0(a) \implies valid(d) = v_0(d)$ is true in the post condition.

Proof. Let $d \in ADR$ be an address such that $d \neq e_0(a)$ and $d \neq e(a)$. We must show $valid(d) = v_0(d)$.

From ④b we have $valid(d) = v_1(d)$.

From ② and ③ because $d \neq e_0(a)$, we have $v_0(d) = v_1(d)$.

Therefore, by combining the above conditions $(d \neq e_0(a))$, we conclude $valid(d) = v_1(d) = v_0(d)$.

□

6.3.4 Function Call

The de-allocation analyser takes a function call at WyIL level as input and checks the properties of each array parameter to determine the flag passed to called function, and to indicate whether input parameter can be freed by the callee. After the function call, the analyser also adds extra code to change run-time de-allocation flags, depending on the aliasing of function return and passed parameter.

To avoid dangling pointers occurring during function call, the analyser uses below rules to decide the flag value:

- *Single flag rule* ensures that each integer typed array has only one de-allocation flag. And our deallocation is acted on the entire array, so all its sub arrays must be reclaimed back to system once the array is freed.

  However, primitive integer or boolean typed parameter does not have the de-allocation flag as they are allocated on stack memory and deleted automatically by the system without any manual deallocation.

- *Single function rule* avoid double freeing problems by restricting each array parameter is only deleted by one function.

6.3.4.1 `RETAIN_DEALLOC` macro

```
1 #define RETAIN_DEALLOC(a, b)
2 {
3   PRE_DEALLOC(a);
4   a := func(b, false); // Do not free 'b' at 'func'
5   a_dealloc := true;// No change to 'b_dealloc'
6 }
```

`RETAIN_DEALLOC(a, b)` can be expanded into as follows:

- `PRE_DEALLOC` macro may be used to empty variable a and reset its flag;

- *func* function does not change or return parameter b. That means, b is read-only and not aliased to the return of *func* function. Therefore, *func* can borrow variable b without a copy and thus does not need to de-allocate b.

- the return value of *func* function is passed and assigned to a, so the flag is delegated to a

Assumption 4 *For a function call $a := func(b)$, we include a precondition*

$$e(a) \neq e(b)$$

to ensure $a_{dealloc}$ and $b_{dealloc}$ both can not be true when variable a and b are aliased to the same memory space before the function call.

Also, we need an extra precondition

$$valid(e(b)) = true$$

to ensure the memory address pointed by variable b is valid and safe to perform the operation. These preconditions prevent `PRE_DEALLOC(a)` from freeing the aliased memory space before the call and avoid null-pointer exception errors.

Assumption 5 *The called function func takes variable b as an argument and its procedure does not make any change to b nor return b, and does not deallocate b.*

$\{valid(e(b)) \wedge (\forall i \in VARS \bullet e(i) = e_0(i)) \wedge (\forall d \in ADR \bullet valid(d) = v_0(d))\}$

$a := func(b,\ false)$;

$\{valid(e(b)) \wedge (\forall i \in VARS \bullet i \not\equiv a \implies e(i) = e_0(i)) \wedge$

$(\forall d \in ADR \bullet d \neq e(a) \implies valid(d) = v_0(d)) \wedge fresh(a)\}$

Theorem 6.6 *If INV holds before `RETAIN_DEALLOC(a, b)` macro, then INV still holds true after the macro, as below Hoare Logic:*

$\{INV \wedge (\forall i \in VARS \bullet e(i) = e_0(i)) \wedge (\forall d \in ADR \bullet valid(d) = v_0(d))$

$\wedge e(a) \neq e(b) \wedge valid(e(b))\}$

`RETAIN_DEALLOC(a, b)`

$\{INV \wedge (\forall i \in VARS \bullet i \not\equiv a \wedge i \not\equiv a_{dealloc} \implies e(i) = e_0(i))$⑥$\wedge$

$(\forall d \in ADR \bullet d \neq e_0(a) \wedge d \neq e(a) \implies valid(d) = v_0(d)) \wedge$

$valid(e(b)) \wedge valid(e(a)) \wedge e(a_{dealloc})\}$

$\{INV \wedge (\forall i \in VARS \bullet e(i) = e_0(i)) \wedge (\forall d \in ADR \bullet valid(d) = v_0(d)) \wedge$
$\quad e(a) \neq e(b) \wedge valid(e(b))\}$①
`PRE_DEALLOC(a);`
$\{INV \wedge (\forall i \in VARS \bullet i \not\equiv a \wedge i \not\equiv a_{dealloc} \implies e(i) = e_0(i))$
$\quad \wedge (\forall d \in ADR \bullet d \neq e_0(a) \implies valid(d) = v_0(d)) \wedge$
$\quad e_0(a) \neq e(b) \wedge valid(e(b))\}$②

$\{INV \wedge valid(e(b)) \wedge (\forall i \in VARS \bullet e(i) = e_1(i)) \wedge$
$\quad (\forall d \in ADR \bullet valid(d) = v_1(d))\}$③

```
a:=func(b, false);
a_dealloc:=true;
```

$\{(\forall i \in VARS \bullet i \not\equiv a \wedge i \not\equiv a_{dealloc} \implies e(i) = e_1(i)) \wedge$
$\quad (\forall d \in ADR \bullet d \neq e(a) \implies valid(d) = v_1(d)) \wedge$
$\quad fresh(a)$④ₐ $\wedge valid(e(a)) \wedge valid(e(b)) \wedge e(a_{dealloc})\}$④

$\{INV \wedge (\forall i \in VARS \bullet i \not\equiv a \wedge i \not\equiv a_{dealloc} \implies e(i) = e_0(i)) \wedge$
$\quad (\forall d \in ADR \bullet d \neq e_0(a) \wedge d \neq e(a) \implies valid(d) = v_0(d)) \wedge$
$\quad valid(e(b)) \wedge valid(e(a)) \wedge e(a_{dealloc})\}$⑤

Listing 6.5: Tableau of `RETAIN_DEALLOC(a, b)` macro

Assumption ④ₐ Show $fresh(a)$ is true in the post condition of the macro. From 5, we know *func* does not return nor de-allocate parameter b, and also the returning result a is not aliased to b or any other variable at caller site. Therefore, we include $fresh(a)$ in the post condition.

Reasoning about ①...⑤ Show Tableau 6.6 is almost the same as `ADD_DEALLOC` macro (see 6.3.3.1). Thus, we can follow the same idea to prove `RETAIN_DEALLOC`.

6.3.4.2 `RESET_DEALLOC` macro

```
#define RESET_DEALLOC(a, b)
{
  PRE_DEALLOC(a);
  a := func(b, false);// Do not free 'b' at 'func'
  if(a != b){
    a_dealloc := true;// 'a' and 'b' are NOT aliased
  }else{
    a_dealloc := b_dealloc;// 'a' and 'b' are aliased
    b_dealloc := false;
  }
}
```

`RESET_DEALLOC(a, b)` can be expanded as follows:

- pre-deallocation macro may empty variable a and reset its flag value;

- the called function *func* does not change parameter b but may pass back b to caller site. So variable b can not be freed by *func* because it would cause dangerous null-pointer de-reference error;
- *func* may or may not return variable b, so the aliasing of a and b at caller site is not certain.

  We discuss the function call with two cases:

  - Case 1: b is returned and aliased to a

    We transfer b's flag to a's flag.
  - Case 2: b is NOT returned and NOT aliased to a

    We assign the flag to variable a.

Assumption 6 *For a function call $a := func(b)$, we include a precondition*

$$e(a) \neq e(b)$$

to ensure variable a and b both can not have true flag when they are aliased to the same memory space before the function call.

Also, we need an extra precondition

$$valid(e(b)) = true$$

to ensure the memory address pointed by variable b is valid and safe to perform the operation.

These preconditions prevent `PRE_DEALLOC(a)` *macro from freeing the aliased memory space before the function call, and avoid null-pointer exception errors.*

Assumption 7 *The called function func takes b as an argument and its procedure does not change b, but may or may not return b so that func does not de-allocate b. We define the behaviour of func as below:*

$\{valid(e(b)) \wedge (\forall i \in VARS \bullet e(i) = e_0(i)) \wedge (\forall d \in ADR \bullet valid(d) = v_0(d))\}$

`a` := `func(b, false)`;

$\{valid(e(a)) \wedge (\forall i \in VARS \bullet i \not\equiv a \implies e(i) = e_0(i)) \wedge$

$(\forall d \in ADR \bullet d \neq e(a) \implies valid(d) = v_0(d)) \wedge (e(a) = e(b) \vee fresh(a))\}$

Theorem 6.7 *If INV holds before* `RESET_DEALLOC(a, b)` *macro, then INV still holds true after the macro, as below Hoare Logic:*

$$\{INV \wedge (\forall i \in VARS \bullet e(i) = e_0(i))$$
$$\wedge (\forall d \in ADR \bullet valid(d) = v_0(d))$$
$$\wedge e(a) \neq e(b) \wedge valid(e(b))\}$$

`RESET_DEALLOC(a, b)`

$$\{INV \wedge (\forall i \in VARS \bullet i \not\equiv a \wedge i \not\equiv a_{dealloc} \wedge i \not\equiv b_{dealloc} \implies e(i) = e_1(i))$$
$$\wedge (\forall d \in ADR \bullet d \neq e_0(a) \wedge d \neq e(a) \implies valid(d) = v_0(d))$$
$$\wedge valid(e(b)) \wedge valid(e(a)) \wedge e(a_{dealloc})\}$$

```
1  {INV ∧ (∀i ∈ VARS • e(i) = e0(i)) ∧ (∀d ∈ ADR • valid(d) = v0(d))∧
2       e(a) ≠ e(b) ∧ valid(e(b))}①
3  PRE_DEALLOC(a);
4  {INV ∧ (∀i ∈ VARS • i ≢ a ∧ i ≢ a_dealloc ⟹ e(i) = e0(i))
5       ∧(∀d ∈ ADR • d ≠ e0(a) ⟹ valid(d) = v0(d))∧
6       e0(a) ≠ e(b) ∧ valid(e(b))}②
7
8  {INV ∧ valid(e(b)) ∧ (∀i ∈ VARS • e(i) = e1(i))∧
9       (∀d ∈ ADR • valid(d) = v1(d))}③
10
11 a:=func(b, false);
12 if(a != b) {
13   a_dealloc := true;
14   {(∀i ∈ VARS • i ≢ a ∧ i ≢ a_dealloc ⟹ e(i) = e1(i))④a∧
15        (∀d ∈ ADR • d ≠ e(a) ⟹ valid(d) = v1(d))④b∧
16        (e(a) ≠ e(b) ∧ fresh(a) ∧ valid(e(a)) ∧ e(a_dealloc)④c}
17
18   {INV⑤a ∧ (∀i ∈ VARS • i ≢ a ∧ i ≢ a_dealloc ⟹ e(i) = e1(i))∧
19        (∀d ∈ ADR • d ≠ e(a) ⟹ valid(d) = v1(d))∧
20        (e(a) ≠ e(b) ∧ valid(e(a)) ∧ e(a_dealloc)}
21 }else{
22   a_dealloc := b_dealloc;
23   b_dealloc := false;
24   {(∀i ∈ VARS • i ≢ a ∧ i ≢ a_dealloc ∧ i ≢ b_dealloc ⟹ e(i) = e1(i))④d∧
25        (∀d ∈ ADR • d ≠ e(a) ⟹ valid(d) = v1(d))④e∧
26        e(a) = e(b) ∧ valid(e(a)) ∧ e(a_dealloc) = e1(b_dealloc) ∧ ¬e(b_dealloc)④f}
27
28   {INV⑤b∧
29        (∀i ∈ VARS • i ≢ a ∧ i ≢ a_dealloc ∧ i ≢ b_dealloc ⟹ e(i) = e1(i))∧
30        (∀d ∈ ADR • d ≠ e(a) ⟹ valid(d) = v1(d))∧
31        e(a) = e(b) ∧ valid(e(a)) ∧ valid(e(b))}
32 }
33
34 {INV⑥a∧
35  (∀i ∈ VARS • i ≢ a ∧ i ≢ a_dealloc ∧ i ≢ b_dealloc ⟹ e(i) = e0(i))⑥b∧
36  (∀d ∈ ADR • d ≠ e0(a) ∧ d ≠ e(a) ⟹ valid(d) = v0(d))⑥c∧
37  valid(e(b)) ∧ valid(e(a)))}
```

Listing 6.6: Tableau of `RESET_DEALLOC(a, b)` macro

Assumption ① Assume $e(a) \neq e(b)$ and $valid(e(b))$ are true in the entry condition of the macro as we justify the precondition at start of an assignment (see Definition 6).

Reasoning about ② Show that $e_0(a) \neq e(b)$ and $valid(e(b))$ are true in post condition of the pre-deallocation macro (refer to Theorem 6.2).

Assumption ③ Define $e_1(i)$ and $v_1(d)$ to store the addresses and validity of variables respectively after `PRE_DEALLOC` macro.

Reasoning about ④a, ④b and ④c Show function return a is not aliased with parameter b. From Assumption 7, we have below conditions in the post state.

- $valid(e(a))$, $e(a) \neq e(b)$ and $fresh(a)$ are included since function return is a valid address and different from parameter b (refer to 6);
- because only variable a and $a_{dealloc}$ are changed, the values of all other variables remain unchanged $\forall i \in VARS \bullet i \not\equiv a \wedge i \not\equiv a_{dealloc} \implies e(i) = e_1(i)$ and
- the validity of all array variables, apart from a, should be the same before *func* function call $\forall d \in ADR \bullet d \neq e(a) \implies valid(d) = v_1(d)$

Reasoning about ④d, ④e and ④f Show function return a is aliased with parameter b. From Assumption 7, we have below conditions in the post state.

- $valid(e(a))$ comes from Assumption 7; $e(a) = e(b)$ is included as a and b are aliased from IF branch;
- because only variable a, $a_{dealloc}$ and $b_{dealloc}$ are changed, the values of all other variables remain unchanged, $\forall i \in VARS \bullet i \not\equiv a \wedge i \not\equiv a_{dealloc} \implies e(i) = e_1(i)$;

- from Assumption 7 and no malloc/free code from line 21 to 22, the validity of all array variables, apart from a, should be the same as before *func* function call $\forall d \in ADR \bullet d \neq e(a) \implies valid(d) = v_1(d)$;
- variable b's flag is transferred to a, so we have $e(a_{dealloc}) = e_1(b_{dealloc})$ in line 21 and $\neg e(b_{dealloc})$ in line 22.

Reasoning about (5a) Show that INV holds at the end of IF branch.

$$INV : \forall i, j \in VARS \bullet inv\_dealloc(i, j)\text{Ⓐ} \wedge \forall i \in ARRVARS \bullet inv\_arr(i)\text{Ⓑ}$$

Proof. [Reasoning Deallocation Invariant Ⓐ]
Let $i, j \in VARS$ be the witness variables. We must show that $\forall i, j \bullet inv\_dealloc(i, j)$ holds true at (5a). Consider the following cases:

- Case 1: i, j includes both a and b

  Given $i \equiv a \wedge j \equiv b$ (or equivalently $j \equiv a \wedge i \equiv b$), the invariant can be rewritten as:

  $$inv\_dealloc(a, b) : (e(a_{dealloc}) \wedge e(b_{dealloc}) \wedge a \equiv b) \implies e(a) \neq e(b)$$

  By $e(a) \neq e(b)$ from (4c), we can conclude $inv\_dealloc(a, b)$ is true at (5a).

- Case 2: i, j includes a but NOT b

  Given $i \equiv a \wedge j \not\equiv b$ (or equivalently $j \equiv a \wedge i \not\equiv b$), the invariant can be rewritten as:

  $$inv\_dealloc(a, j) : e(a_{dealloc}) \wedge e(j_{dealloc}) \wedge j \not\equiv a \implies e(j) \neq e(a)$$

  Assume that all the preconditions of $inv\_dealloc(a, j)$ are true, including $j \not\equiv a$. Since we get $fresh(a)$ at (4c) and $j \not\equiv a$ we have $e(j) \neq e(a)$. Therefore, we can conclude $inv\_dealloc(a, j)$ is true at (5a)

- Case 3: i, j does NOT include a

  Given $j \not\equiv a$ (or equivalently $i \not\equiv a$), the invariant is as below:

  $$inv\_dealloc(b, j) : e(b_{dealloc}) \wedge e(j_{dealloc}) \wedge b \not\equiv j \implies e(b) \neq e(j)$$

Because $b \not\equiv a$ and $j \not\equiv a$ from the given assumptions, and only a and $a_{dealloc}$ are changed, we know variable j and $j_{dealloc}$ and b and $b_{dealloc}$ therefore stay unchanged at (5a).

Since $inv\_dealloc(b, j)$ was true at (3), we have $inv\_dealloc(b, j)$ in at (5a).

□

Proof. [Reasoning Array Invariant Ⓑ]

We must show $\forall i \in ARRVARS \bullet e(i_{dealloc}) \implies valid(e(i))$ holds true in the post-state.

Let $i \in ARRVARS$ such that $e(i_{dealloc})$ is true in the post-state and we have to show $valid(e(i))$.

- Case 1: $i \not\equiv a$

  We know $e(i) \neq e(a)$ because of $fresh(a)$ from (4c).

  From $inv\_arr(i)$ at (3), we have

  $$e_1(i_{dealloc}) \implies v_1(e_1(i))$$

  From (4a), we have $e(i) = e_1(i)$ because $i \not\equiv a \wedge i \not\equiv a_{dealloc}$.

  Since $e(i_{dealloc})$ is assumed to be true from given assumption, $e_1(i_{dealloc})$ is true in the post state because $i \not\equiv a$. So we get

  $$v_1(e_1(i)) = true$$

  From (4b), we have $valid(e(i)) = v_1(e(i))$ because $e(i) \neq e(a)$

  Therefore, the validity must remain unchanged in the post state.

  $$valid(e(i)) = v_1(e(i)) = v_1(e_1(i)) = true$$

- Case 2: $i \equiv a$

  $inv\_arr(a) : e(a_{dealloc}) \implies valid(e(a))$ holds true because we have $valid(e(a))$ in (4c).

□

Reasoning about ⑤b Show that INV holds at the end of Else branch.

$$INV : \forall i, j \in VARS \bullet inv\_dealloc(i, j)\text{Ⓐ} \wedge \forall i \in ARRVARS \bullet inv\_arr(i)\text{Ⓑ}$$

Proof. [Reasoning Deallocation Invariant Ⓐ]
Let $i, j \in VARS$ be the witness variables. We must show $\forall i, j \bullet inv\_dealloc(i, j)$: $e(i_{dealloc}) \wedge e(j_{dealloc}) \wedge i \not\equiv j \implies e(i) \neq e(j)$ holds true at ⑤b.

As $inv\_dealloc(i, j)$ is symmetric, we can swap variable i and j without breaking the invariant, so $inv\_dealloc(i, j) \iff inv\_dealloc(j, i)$ and the reasoning just needs to consider three cases:

- Case 1: i, j includes b

  Given $j \equiv b$ (or equivalently $i \equiv b$), the invariant can be rewritten as:

  $$inv\_dealloc(i, b) : (e(i_{dealloc}) \wedge e(b_{dealloc}) \wedge i \not\equiv b) \implies e(i) \neq e(b)$$

  Since $e(b_{dealloc})$ is false by ④f, $inv\_dealloc(i, b)$ holds true after the macro.

- Case 2: i, j includes a but NOT b

  Given $i \equiv a \wedge j \not\equiv b$ (or equivalently $j \equiv a \wedge i \not\equiv b$), the invariant can be rewritten as:

  $$inv\_dealloc(a, j) : (e(a_{dealloc}) \wedge e(j_{dealloc}) \wedge j \not\equiv a) \implies e(a) \neq e(j)$$

  Because of $e(a_{dealloc}) = e_1(b_{dealloc})$ from ④f, this is the same as

  $$inv\_dealloc(a, j) : (e_1(b_{dealloc}) \wedge e(j_{dealloc}) \wedge j \not\equiv a) \implies e(a) \neq e(j)$$

  where $j \not\equiv b$ by the assumption of this case.

  Assume the preconditions of above implication hold in the post-state. We need to show $e(a) \neq e(j)$:

  - From $e_1(b_{dealloc})$ we conclude $e(b_{dealloc})$ was true at ③.
  - From $e(j_{dealloc})$ we conclude $e(j_{dealloc})$ was true at ③, because $j \not\equiv a, b$ and the macro only changes $a_{dealloc}$ and $b_{dealloc}$ (and a).

Because $j \not\equiv b$ and $inv\_dealloc(b, j)$ was true at ③, we get $e_1(b) \neq e_1(j)$. Finally, by using $e(a) = e(b) = e_1(b)$ from ④f and $e(j) = e_1(j)$ from ④d, because $j \not\equiv a$ and $j \not\equiv b$, we have $e(a) \neq e(j)$ in the post-state. So $inv\_dealloc(b, j)$ is true at ⑤b.

- Case 3: i, j does NOT include a or b

  The instructions of reset macro do not change anything, except for $a_{dealloc}$ and $b_{dealloc}$ and a. That means, $e(i_{dealloc}) = e_1(i_{dealloc})$ and $e(i) = e_1(i)$ for $i \not\equiv a \wedge i \not\equiv a_{dealloc} \wedge i \not\equiv b_{dealloc}$. Since $inv\_dealloc(i, j)$ was true in ③, we therefore can conclude $inv\_dealloc(i, j)$ is still true at ⑤b.

□

Proof. [Reasoning Array Invariant Ⓑ]

We must show $inv\_arr(i) : \forall i \in ARRVARS \bullet e(i_{dealloc}) \implies valid(e(i))$ holds true in the post-state.

Let $i \in ARRVARS$ such that $e(i_{dealloc})$ is true at ⑤b and we have to show $valid(e(i))$.

- Case 1: $e(i) \neq e(a)$ (thus $i \not\equiv a$)

  We get

  - $valid(e(i)) = v_1(e(i))$ from ④e because $e(i) \neq e(a)$
  - $e(i) = e_1(i)$ from ④d because $i \not\equiv a$
  - $e_1(i_{dealloc}) = e(i_{dealloc}) = true$ because $i \not\equiv a$ from assumption and $i \not\equiv b$ from $e(i_{dealloc}) = true$ and $e(b_{dealloc}) = false$.
  - $inv\_arr(i)$ at ⑤b: $e_1(i_{dealloc}) \implies v_1(e_1(i))$, so $v_1(e_1(i)) = true$.

  With above, we conclude

  $$valid(e(i)) = v_1(e(i)) = v_1(e_1(i)) = true$$

- Case 2: $e(i) = e(a)$

  We have $valid(e(i)) = valid(e(a)) = true$ from ④f

□

Reasoning about (6a) Show *INV* holds true in the post condition of reset macro as *INV* is true at both if branch (see (5a)) and else branches (see (5b)).

Reasoning about (6b) Show $\forall i \in VARS \bullet i \not\equiv a \wedge i \not\equiv a_{dealloc} \wedge i \not\equiv b_{dealloc} \implies e(i) = e_0(i)$ is true in the post condition.

Proof. Let $i \in VARS$ be a variable such that $i \not\equiv a$ and $i \not\equiv a_{dealloc}$ and $i \not\equiv b_{dealloc}$. We must show $e(i) = e_0(i)$.

From (4a) and (4d) because $i \not\equiv a \wedge i \not\equiv a_{dealloc} \wedge i \not\equiv b_{dealloc}$, we have $e(i) = e_1(i)$.

From (2) and (3) because $i \not\equiv a \wedge i \not\equiv a_{dealloc}$, we have $e_0(i) = e_1(i)$.

Therefore, by combining the above conditions $i \not\equiv a \wedge i \not\equiv a_{dealloc} \wedge i \not\equiv b_{dealloc}$, we conclude $e(i) = e_0(i)$.

□

Reasoning about (6c) Show $\forall d \in ADR \bullet d \neq e_0(a) \wedge d \neq e(a) \implies valid(d) = v_0(d)$ is true in the post condition.

Proof. Let $d \in ADR$ be an address such that $d \neq e_0(a)$ and $d \neq e(a)$. We must show $valid(d) = v_0(d)$.

From (4b) and (4e) because $d \neq e(a)$, we have $valid(d) = v_1(d)$.

From (2) and (3) because $d \neq e_0(a)$, we have $v_0(d) = v_1(d)$.

Therefore, by combining the above conditions $(d \neq e_0(a) \wedge d \neq e(a))$, we conclude $valid(d) = v_1(d) = v_0(d)$.

□

6.3.4.3 `CALLER_DEALLOC` macro

```
1  #define CALLER_DEALLOC(a, b)
2  {
3    PRE_DEALLOC(a);
4    tmp_dealloc := false;// Do not free copied 'b' at 'func'
5    a := func(tmp := copy(b), tmp_dealloc);
6    if (a != tmp) {// Possible memory leak on 'tmp'
7      free(tmp);
8    }
9    a_dealloc := true;
10 }
```

`CALLER_DEALLOC(a, b)` can be expanded into below:

- `PRE_DEALLOC(a)` may be used to empty variable a and reset its flag;
- Parameter b may be changed by the called function *func*. Since b's value will be used after the call, b is copied to *tmp* first and then passed to *func*. In doing so, we can eliminate the side effects from function calls.
- Variable *tmp* is or may not be returned by *func*, so *tmp* will not be deallocated by the called function *func*. If *tmp* and a are not aliased and different, then function *func* does not return variable *tmp* and thus *tmp* is the extra copy and can be safely deleted at caller site.
- Variable a is assigned with true flag to give it the responsible to free the allocated memory space.

Assumption 8 *For a function call* $a := func(copy(b))$*, we include a precondition*

$$e(a) \neq e(b)$$

to ensure variable a and b both can not have true flag when they are aliased to the same memory space before the function call.

Also, we need an extra precondition

$$valid(e(b)) = true$$

to ensure the memory address pointed by variable b is valid and safe to perform the operation.

These preconditions stop `PRE_DEALLOC(a)` *macro freeing the aliased memory space of variable b before the function call, and avoid segmentation errors when copying variable b.*

We also include an assumption

$$a \not\equiv tmp$$

to ensure tmp and a have different variable names. By adding this assumption, we can be sure variable tmp does not duplicate the name of variable a so avoids

potential variable shadowing, which uses the same variable name in different scopes of the macro.

Assumption $a \not\equiv tmp$ *stays consistently within the macro because we use a naming rule to make variable tmp distinct from variable a. This assumption is found by automatic prover Boogie (see Section 6.4).*

Assumption 9 *The called function func takes tmp as an argument, and may change tmp but may return tmp to the caller site. We define the behaviour of function func as below:*

$$
\begin{aligned}
&\{a \not\equiv tmp \wedge valid(e(tmp)) \\
&\qquad \wedge (\forall i \in VARS \bullet e(i) = e_0(i)) \\
&\qquad \wedge (\forall d \in ADR \bullet valid(d) = v_0(d))\} \\
&\texttt{a} := \texttt{func}(\texttt{tmp},\ \texttt{false}); \\
&\{a \not\equiv tmp \wedge (fresh(a) \vee e(a) = e(tmp)) \wedge valid(e(a)) \\
&\qquad \wedge (\forall i \in VARS \bullet i \not\equiv a \implies e(i) = e_0(i)) \\
&\qquad \wedge (\forall d \in ADR \bullet d \neq e(a) \implies valid(d) = v_0(d))\}
\end{aligned}
$$

Theorem 6.8 *If INV holds before* `CALLER_DEALLOC(a, b)` *macro, then INV still holds true after the macro, as below Hoare Logic:*

$$
\begin{aligned}
&\{a \not\equiv tmp \wedge INV \\
&\qquad \wedge (\forall i \in VARS \bullet e(i) = e_0(i)) \\
&\qquad \wedge (\forall d \in ADR \bullet valid(d) = v_0(d))\} \\
&\qquad \wedge e(a) \neq e(b) \wedge valid(e(b))
\end{aligned}
$$

`CALLER_DEALLOC(a, b)`

$$
\begin{aligned}
&\{a \not\equiv tmp \wedge INV \\
&\qquad (\forall i \in VARS \bullet i \not\equiv a \wedge i \not\equiv a_{dealloc} \wedge i \not\equiv tmp \wedge i \not\equiv tmp_{dealloc} \implies e(i) = e_0(i)) \\
&\qquad \wedge (\forall d \in ADR \bullet d \neq e_0(a) \wedge d \neq e(a) \wedge d \neq e(tmp) \implies valid(d) = v_0(d)) \\
&\qquad \wedge valid(e(a)) \wedge e(a_{dealloc})\}
\end{aligned}
$$

```
1  {a ≢ tmp ∧ INV∧
2     (∀i ∈ VARS • e(i) = e0(i)) ∧ (∀d ∈ ADR • valid(d) = v0(d))∧
3     e(a) ≠ e(b) ∧ valid(e(b))}①
4  PRE_DEALLOC(a);
5  {a ≢ tmp ∧ INV ∧ e0(a) ≠ e(b) ∧ valid(e(b))∧
6     (∀i ∈ VARS • i ≢ a ∧ i ≢ a_dealloc ⟹ e(i) = e0(i))
7     ∧(∀d ∈ ADR • d ≠ e0(a) ⟹ valid(d) = v0(d))}②
8
9  {a ≢ tmp ∧ INV ∧ valid(e(b)) ∧ (∀i ∈ VARS • e(i) = e1(i))∧
10    (∀d ∈ ADR • valid(d) = v1(d))}③
11 tmp := copy(b);
12 tmp_dealloc:=false;
13 {a ≢ tmp ∧ valid(e(tmp)) ∧ fresh(tmp) ∧ ¬e(tmp_dealloc)∧
14    (∀i ∈ VARS • i ≢ tmp ∧ i ≢ tmp_dealloc ⟹ e(i) = e1(i))∧
15    (∀d ∈ ADR • d ≠ e(tmp) ⟹ valid(d) = v1(d))}④
16
17 {a ≢ tmp ∧ valid(e(tmp)) ∧ fresh(tmp) ∧ ¬e(tmp_dealloc)∧
18    (∀i ∈ VARS • e(i) = e2(i))∧
19    (∀d ∈ ADR • valid(d) = v2(d))}⑤
20 a := func(tmp, tmp_dealloc);
21 {a ≢ tmp ∧ (e(a) = e(tmp) ∨ fresh(a)) ∧ valid(e(a)) ∧ valid(e(tmp))∧
22    ¬e(tmp_dealloc) ∧ (∀i ∈ VARS • i ≢ a ⟹ e(i) = e2(i))∧
23    (∀d ∈ ADR • d ≠ e(a) ⟹ valid(d) = v2(d))}⑥
24 if (a != tmp) {
25    {a ≢ tmp ∧ (e(a) ≠ e(tmp) ∧ fresh(a) ∧ valid(e(a)) ∧ ¬e(tmp_dealloc)∧
26       (∀i ∈ VARS • e(i) = e3(i))∧
27       (∀d ∈ ADR • valid(d) = v3(d))}⑦a
28
29    free(tmp);
30
31    {a ≢ tmp ∧ (e(a) ≠ e(tmp) ∧ fresh(a) ∧ valid(e(a)) ∧ ¬e(tmp_dealloc)∧
32       (∀i ∈ VARS • i ≢ a ⟹ e(i) = e2(i))∧
33       (∀d ∈ ADR • d ≠ e(a) ∧ d ≠ e(tmp) ⟹ valid(d) = v2(d))}⑦b
34
35 }else{
36    // Do nothing
37
38    {a ≢ tmp ∧ e(a) = e(tmp) ∧ valid(e(a)) ∧ valid(e(tmp)) ∧ ¬e(tmp_dealloc)∧
39       (∀i ∈ VARS • i ≢ a ⟹ e(i) = e2(i))∧
40       (∀d ∈ ADR • d ≠ e(a) ⟹ valid(d) = v2(d))}⑦c
41
42 }
43
44 {a ≢ tmp ∧ valid(e(a)) ∧ ¬e(tmp_dealloc)∧
45    (∀i ∈ VARS • i ≢ a ∧ i ≢ tmp ∧ i ≢ tmp_dealloc ⟹ e(i) = e1(i))∧
46    (∀d ∈ ADR • d ≠ e(a) ∧ d ≠ e(tmp) ⟹ valid(d) = v1(d))∧
47    (∀i ∈ ARRVARS • i ≢ a ∧ i ≢ tmp ⟹ e(i) ≠ e(a))}⑧
48
49 a_dealloc := true;
50 {a ≢ tmp ∧ e(a_dealloc) ∧ valid(e(a)) ∧ ¬e(tmp_dealloc)∧
51  (∀i ∈ VARS • i ≢ a ∧ i ≢ tmp ∧ i ≢ a_dealloc ∧ i ≢ tmp_dealloc
52                ⟹ e(i) = e1(i))⑨a∧
53  (∀d ∈ ADR • d ≠ e(a) ∧ d ≠ e(tmp) ⟹ valid(d) = v1(d))⑨b∧
54  (∀i ∈ ARRVARS • i ≢ a ∧ i ≢ tmp ⟹ e(i) ≠ e(a))⑨c}⑨
55
56 {a ≢ tmp ∧ INV⑩a ∧ valid(e(a)) ∧ e(a_dealloc)∧
57  (∀i ∈ VARS • i ≢ a ∧ i ≢ a_dealloc ∧ i ≢ tmp ∧ i ≢ tmp_dealloc
58                ⟹ e(i) = e0(i))⑩b∧
   (∀d ∈ ADR • d ≠ e0(a) ∧ d ≠ e(a) ∧ d ≠ e(tmp) ⟹ valid(d) = v0(d))⑩c
```

Listing 6.7: Tableau of `CALLER_DEALLOC(a, b)` macro

Assumption ① Assume $e(a) \neq e(b)$ and $valid(e(b))$ are true in the entry condition of the macro as we provide evidence for the precondition at start of Assumption 8.

Reasoning about ② $e_0(a) \neq e(b)$ and $valid(e(b))$ are true in post condition of the pre-deallocation macro (refer to Theorem 6.2).

Assumption ③ Define $e_1(i)$ and $v_1(d)$ to store the variables addresses and their validity respectively after `PRE_DEALLOC` macro.

Reasoning about ④ Show $valid(e(tmp))$ and $fresh(tmp)$ and $\neg e(tmp_{dealloc})$ are true in the post-condition.

$valid(e(tmp)) \wedge fresh(tmp)$ is included because we make a fresh copy of variable b to tmp in line 11, and from line 12 we assign a false flag to $tmp\_dealloc$.

Assumption ⑤ Define $e_2(i)$ as the value of variable i at line 17 before the function call. And we also define $v_2(d)$ as the validity of address d at line 17. And $a \not\equiv tmp$ is included from given Assumption 8 to ensure actual parameter tmp and function return a are not aliased before the call, so that we will not introduce side effects to called function.

Assumption ⑥ $(e(a) = e(tmp) \vee fresh(a)) \wedge valid(e(a)$ holds true because of function behaviour (see Assumption 9). From line 17 $valid(e(tmp)) \wedge \neg e(tmp_{dealloc})$ is included in the post state.

Assumption ⑦a Define $e_3(i)$ to store $e_2(i)$ values of variable i before free statement at line 29. Also, we define $v_3(d)$ to store $v_2(i)$ validity of address d.

Reasoning about (7b) Show

$$\{a \not\equiv tmp \wedge e(a) \neq e(tmp) \wedge fresh(a) \wedge valid(e(a)) \wedge \neg e(tmp_{dealloc})$$
$$\forall i \in VARS \bullet i \not\equiv a \implies e(i) = e_2(i) \wedge$$
$$\forall d \in ADR \bullet d \neq e(a) \wedge d \neq e(tmp) \implies valid(d) = v_2(d)\}$$

Proof. From (7a) to (7b) all the other variables, except for tmp, are unchanged and thus we can include $e(a) \neq e(tmp) \wedge fresh(a) \wedge valid(e(a)) \wedge \neg e(tmp_{dealloc})$ in the post condition. And $a \not\equiv tmp$ is included to the post state because of given assumption.

By combining (6) and (7a), we get $e(i) = e_3(i) = e_2(i)$ except for $i \not\equiv a$.

We have $\neg valid(e(tmp))$ in the post state of `free(tmp)` statement, so we can get $v_3(d) = v_2(d)$ for $d \neq e(tmp)$. Also, we have $v_2(d) = v_3(d)$ for $d \neq e(a)$ from (6) and (7a).

We can write the validity as follows:

$$valid(d) = v_3(d) = v_2(d) \text{ for } d \neq e(a) \wedge d \neq e(tmp)$$

□

Reasoning about (7c) Show

$$\{a \not\equiv tmp \wedge e(a) = e(tmp) \wedge valid(e(a)) \wedge valid(e(tmp)) \wedge \neg e(tmp_{dealloc})$$
$$\forall i \in VARS \bullet i \not\equiv a \implies e(i) = e_2(i) \wedge$$
$$\forall d \in VARS \bullet d \neq e(a) \implies valid(d) = v_2(d)\}$$

Proof. Because it is in ELSE branch, we have $e(a) = e(tmp)$ in the post condition. And we can include $valid(e(a)) \wedge valid(e(tmp)) \wedge \neg e(tmp_{dealloc})$ as those are not changed.

We can just repeat $valid(d) = v_2(d)$ for $d \neq e(a)$ and $e(i) = e_2(i)$ for $i \not\equiv a$ from (6) because they have no change in ELSE branch. □

Reasoning about ⑧ We have the post-condition of IF branch:

$$\{a \not\equiv tmp \wedge e(a) \neq e(tmp) \wedge fresh(a) \wedge valid(e(a)) \wedge \neq e(tmp_{dealloc})$$
$$\forall i \in VARS \bullet i \not\equiv a \implies e(i) = e_2(i) \wedge$$
$$\forall d \in VARS \bullet d \neq e(a) \wedge d \neq e(tmp) \implies valid(d) = v_2(d)\}\text{⑦b}$$

and ELSE branch :

$$\{a \not\equiv tmp \wedge e(a) = e(tmp) \wedge valid(e(a)) \wedge valid(e(tmp)) \wedge \neq e(tmp_{dealloc})$$
$$\forall i \in VARS \bullet i \not\equiv a \implies e(i) = e_2(i) \wedge$$
$$\forall d \in VARS \bullet d \neq e(a) \implies valid(d) = v_2(d)\}\text{⑦c}$$

In the post state of IF and ELSE branches, we get ⑦b $\vee$ ⑦c. So we must show this implies:

$$\{a \not\equiv tmp \wedge valid(e(a)) \wedge \neg e(tmp_{dealloc}) \wedge$$
$$(\forall i \in VARS \bullet i \not\equiv a \implies e(i) = e_2(i)) \wedge$$
$$(\forall d \in ADR \bullet d \neq e(a) \wedge d \neq e(tmp) \implies valid(d) = v_2(d)) \wedge$$
$$(\forall i \in ARRVARS \bullet i \not\equiv a \wedge i \not\equiv tmp \implies e(i) \neq e(a))\}\text{⑧}$$

From ⑦b and ⑦c, we have $valid(e(a)) \wedge \neg e(tmp_{dealloc})$ in the post state.

Also, we have common $\forall i \in VARS \bullet i \not\equiv a \implies e(i) = e_2(i))$.

By combining the validity at ⑦b and ⑦c, we have $valid(d) = v_2(d)$ for addresses $d \not\equiv e(a) \wedge d \not\equiv e(tmp)$.

Consider ⑦b and ⑦c separately. We can show:

$$\forall i \in ARRVARS \bullet i \not\equiv a \wedge i \not\equiv tmp \implies e(i) \neq e(a)$$

Proof. Let $i \in ARRVARS$ such that $i \not\equiv a$ and $i \not\equiv tmp$.

- Case ⑦b

  We have $fresh(a)$ at ⑦b:

  $$fresh(a) : \forall i \in ARRVARS \bullet i \not\equiv a \implies e(i) \neq e(a)$$

  And because $a \not\equiv tmp$, we get

  $$\forall i \in ARRVARS \bullet i \not\equiv tmp \wedge i \not\equiv a \implies e(i) \neq e(a)$$

- Case (7c)

  $fresh(tmp)$ at (5) means:

  $$\forall i \in ARRVARS \bullet i \not\equiv tmp \implies e_2(i) \neq e_2(tmp)\text{Ⓐ}$$

  From (7c), we have

  $$e(a) = e(tmp)\text{Ⓑ}$$
  $$(a \not\equiv tmp) \wedge (\forall i \in VARS \bullet i \not\equiv a \implies e(i) = e_2(i))\text{Ⓒ}$$

  Proof. Let $i \in ARRVARS$ such that $i \not\equiv a$ and $i \not\equiv tmp$.

  By Ⓒ, $e(i) = e_2(i)$ and also $e(tmp) = e_2(tmp)$ because $tmp \not\equiv a$.

  By Ⓐ, $e_2(i) \neq e_2(tmp)$

  Therefore,

  $$e(i) \overset{\text{Ⓒ}}{=\!=} e_2(i) \overset{\text{Ⓐ}}{\neq} e_2(tmp) \overset{\text{Ⓒ}}{=\!=} e(tmp) \overset{\text{Ⓑ}}{=\!=} e(a)$$

  Since i was chosen arbitrarily:

  $$\forall i \in ARRVARS \bullet i \not\equiv tmp \wedge i \not\equiv a \implies e(i) \neq e(a)$$

□

□

Now we can show (*) implies (8):

$$\{a \not\equiv tmp \wedge valid(e(a)) \wedge \neg e(tmp_{dealloc}) \wedge$$
$$(\forall i \in VARS \bullet i \not\equiv a \implies e(i) = e_2(i)) \wedge$$
$$(\forall d \in ADR \bullet d \neq e(a) \wedge d \neq e(tmp) \implies valid(d) = v_2(d)) \wedge$$
$$(\forall i \in ARRVARS \bullet i \not\equiv a \wedge i \not\equiv tmp \implies e(i) \neq e(a))\}\text{(*)}$$
$$\implies \{a \not\equiv tmp \wedge valid(e(a)) \wedge \neg e(tmp_{dealloc}) \wedge$$
$$(\forall i \in VARS \bullet i \not\equiv a \wedge i \not\equiv tmp \wedge i \not\equiv tmp_{dealloc} \implies e(i) = e_1(i)) \wedge$$
$$(\forall d \in ADR \bullet d \neq e(a) \wedge d \neq e(tmp) \implies valid(d) = v_1(d)) \wedge$$
$$\forall i \in ARRVARS \bullet i \not\equiv a \wedge i \not\equiv tmp \implies e(i) \neq e(a)\}\text{(8)}$$

Proof. Let $i \in VARS$ such that $i \not\equiv a$ and $i \not\equiv tmp$ and $i \not\equiv tmp_{dealloc}$. Then we have $e_1(i) = e_2(i)$ because $i \not\equiv tmp \wedge i \not\equiv tmp_{dealloc}$ at ④, and $e(i) = e_2(i)$ because $i \not\equiv a$ at ⊛. Therefore, we have:

$$\forall i \in VARS \bullet i \not\equiv a \wedge i \not\equiv tmp \wedge i \not\equiv tmp_{dealloc} \implies e(i) = e_2(i) = e_1(i)$$

Let $d \neq e(a)$ and $d \neq e(tmp)$. We must show $valid(d) = v_1(d)$ is true in the post state.

By ⑦c, we have $valid(d) = v_2(d)$...(i)

By ④ + ⑤, we have $d \neq e_2(tmp) \implies v_2(d) = v_1(d)$. By ⑦c, $\forall i \in VARS \bullet i \not\equiv a \implies e(i) = e_2(i)$, because $tmp \not\equiv a$, we have $e(tmp) = e_2(tmp)$. For $d \neq (e(tmp) = e_2(tmp))$ (means $d \neq e_2(tmp)$) we get $v_2(d) = v_1(d)$...(ii)

By combining (i) and (ii), for $d \neq e(a) \wedge d \neq e(tmp)$ we have

$$valid(d) = v_2(d) = v_1(d)$$

The other conditions are unchanged so can be moved to ⑧ as we do not include any extra statement to change any value. □

Assumption ⑨ $e(a_{dealloc})$ is included in the post condition because the assignment changes $a_{dealloc}$ value. In addition, we include $i \not\equiv a_{dealloc}$ to the predicate of $e(i) = e_1(i)$ to reflect this change.

Reasoning about ⑩a Show that INV holds at the end of macro.

$$INV : \forall i, j \in VARS \bullet inv\_dealloc(i, j)Ⓐ \wedge \forall i \in ARRVARS \bullet inv\_arr(i)Ⓑ$$

By ⑨, we assume $a \not\equiv tmp$ in the post state.

Proof. [Reasoning Deallocation Invariant Ⓐ]

We must show $\forall i, j \bullet inv\_dealloc(i, j) : e(i_{dealloc}) \wedge e(j_{dealloc}) \wedge i \not\equiv j \implies e(i) \neq e(j)$ holds true at ⑩a.

Let $i, j \in ARRVARS$ such that $e(i_{dealloc})$ and $e(j_{dealloc})$ and $i \not\equiv j$.

Then $i \not\equiv tmp$ and $j \not\equiv tmp$ because $e(tmp_{dealloc})$ is false at ⑨.

As $inv\_dealloc(i,\ j)$ is symmetric, we can swap variable i and j so $inv\_dealloc(i,j) \iff inv\_dealloc(j,i)$ and the reasoning just needs to consider two cases:

- Case 1: i, j includes a

  Given $i \equiv a$ (or equivalently $j \equiv a$), the invariant can be rewritten as:

  $$inv\_dealloc(a,j) : (e(a_{dealloc}) \wedge e(j_{dealloc}) \wedge a \not\equiv j) \implies e(a) \neq e(j)$$

  From ⑨c, we have $j \not\equiv a$ and $j \not\equiv tmp$ implies $e(j) \neq e(a) = e(i)$, which makes $inv\_dealloc(a,j)$ true in the post state.

- Case 2: i, j does NOT include a

  Given $i \not\equiv a$ and $j \not\equiv a$ and since $i \not\equiv tmp \wedge j \not\equiv tmp$, we have $e(i) = e_1(i)$ and $e(j) = e_1(j)$ from ⑨a.

  Because invariant INV holds at ③, $inv\_dealloc(i,j)$ is true for e_1 and thus we get

  $$e(i) = e_1(i) \neq e_1(j) = e(j)$$

□

Proof. [Reasoning Array Invariant Ⓑ]
We must show $\forall i \in ARRVARS \bullet e(i_{dealloc}) \implies valid(e(i))$ holds true in the post-state.

Let $i \in ARRVARS$ such that $e(i_{dealloc})$ is true in the post-state, and we have to show $valid(e(i))$.

- Case 1: $e(i) = e(a)$

  $e(a_{dealloc}) \implies valid(e(a))$ holds true, because $valid(e(a))$ at ⑨.

- Case 2: $e(i) = e(tmp)$

  $e(tmp_{dealloc}) \implies valid(e(tmp))$ is true because $\neg e(tmp_{dealloc})$ at ⑨.

- Case 3: $e(i) \neq e(a)$ and $e(i) \neq e(tmp)$ (implies $i \not\equiv a$ and $i \not\equiv tmp$)

From $inv\_arr(i)$ at ③, we have $e_1(i_{dealloc}) \implies v_1(e_1(i))$

From ⑨a, we have $e(i) = e_1(i)$ because $i \not\equiv a \wedge i \not\equiv tmp \wedge i \not\equiv a_{dealloc} \wedge i \not\equiv tmp_{dealloc}$.

Since $e(i_{dealloc})$ is assumed to be true, $e_1(i_{dealloc})$ is true in the post state because $i \not\equiv a$ and $i \not\equiv tmp$ by ⑨a. So we get

$$v_1(e_1(i)) = true$$

From ⑨b, we have $valid(e(i)) = v_1(e(i))$ because $e(i) \neq e(a) \wedge e(i) \neq e(tmp)$.

Therefore, the validity must remain unchanged as ③.

$$valid(e(i)) = v_1(e(i)) = v_1(e_1(i)) = true$$

□

Reasoning about ⑩b Show $\forall i \in VARS \bullet i \not\equiv a \wedge i \not\equiv a_{dealloc} \wedge i \not\equiv tmp \wedge i \not\equiv tmp_{dealloc} \implies e(i) = e_0(i)$ is true in the post condition.

Proof. Let $i \in VARS$ be a variable such that $i \not\equiv a$, $i \not\equiv a_{dealloc}$, $i \not\equiv tmp$ and $i \not\equiv tmp_{dealloc}$. We must show $e(i) = e_0(i)$.

From ⑨a because $i \not\equiv a \wedge i \not\equiv tmp \wedge i \not\equiv a_{dealloc} \wedge i \not\equiv tmp_{dealloc}$, we have $e(i) = e_1$... (a)

From ③ and ②, because $i \not\equiv a$ and $i \not\equiv a_{dealloc}$, we get $e_1(i) = e_0$... (b)

By combining (a) and (b) with the predicate $i \not\equiv a$, $i \not\equiv a_{dealloc}$, $i \not\equiv tmp$ and $i \not\equiv tmp_{dealloc}$ we can therefore conclude

$$e(i) = e_1(i) = e_0(i)$$

□

Reasoning about ⑩c Show $\forall d \in ADR \bullet d \neq e_0(a) \wedge d \neq e(a) \wedge d \neq e(tmp) \implies valid(d) = v_0(d)$ is true in the post condition.

Proof. Let $d \in ADR$ be an address such that $d \neq e_0(a)$ and $d \neq e(a)$ and $d \neq e(tmp)$. We must show $valid(d) = v_0(d)$.

From ⑨b because $d \neq e(a) \wedge d \neq e(tmp)$, we have $valid(d) = v_1(d)$... (a)

From ③ and ②, we get $v_1(d) = v_0(d)$ because $d \neq e_0(a)$... (b)

By combining (a) and (b), because $d \neq e(a) \wedge d \neq e(tmp) \wedge d \neq e_0(a)$, we can therefore conclude

$$valid(d) = v_1(d) = v_0(d)$$

□

6.3.4.4 `CALLEE_DEALLOC` macro

```
1 #define CALLEE_DEALLOC(a, b)
2 {
3   PRE_DEALLOC(a);
4   tmp := copy(b);
5   a := func(tmp, true);// Free copied 'b' at 'func'
6   tmp_dealloc = false;
7   a_dealloc := true;// No change to 'b_dealloc'
8 }
```

`CALLEE_DEALLOC(a, b)` can be expanded into below:

- pre-deallocation macro may be used to empty variable a and reset its flag value;
- parameter b may be changed by the called function *func* and its value will be used afterwards. To ensure the immutable values in Whiley functional programming, the parameter b is copied to *tmp* first and passed to *func*, so that we can eliminate the side effects from function calls.

  Since *tmp* is not returned by *func*, it can be de-allocated safely by *func* to avoid the memory leaks.
- *func* does not return variable b, so a and b are not aliased at caller site.

Assumption 10 *For a function call $a := func(copy(b))$, we include a pre-condition*

$$e(a) \neq e(b)$$

to ensure variable a and b both can not have true flag when they are aliased to the same memory space before the function call. Also, we need an extra precondition

$$valid(e(b)) = true$$

to ensure the memory address pointed by variable b is valid and safe to perform the operation.

We also include an assumption

$$a \not\equiv tmp$$

to ensure tmp and a have different variable names. By adding this assumption, we can be sure there is no potential variable shadowing, so we can safely assign the value to deallocation run-time flag, such as a_dealloc and tmp_dealloc.

Assumption 11 *The called function func takes tmp as an argument and its procedure may or may not change tmp, but does not return tmp and de-allocates tmp. We define the behaviour of func as below:*

$$\begin{aligned}
&\{a \not\equiv tmp \wedge valid(tmp) \\
&\qquad \wedge (\forall i \in VARS \bullet e(i) = e_0(i)) \\
&\qquad \wedge (\forall d \in ADR \bullet valid(d) = v_0(d))\} \\
&\texttt{a} := \texttt{func}(\texttt{tmp},\ \texttt{true}); \\
&\{a \not\equiv tmp \\
&\qquad \wedge (\forall i \in VARS \bullet i \not\equiv a \implies e(i) = e_0(i)) \\
&\qquad \wedge (\forall d \in ADR \bullet d \neq e(a) \implies valid(d) = v_0(d)) \\
&\qquad \wedge fresh(a) \wedge \neg valid(e(tmp)) \wedge valid(e(a))\}
\end{aligned}$$

Theorem 6.9 *If INV holds before* `CALLEE_DEALLOC(a, b)` *macro, then INV still holds true after the macro, as below Hoare Logic:*

$\{a \not\equiv tmp \wedge INV$

$\quad \wedge (\forall i \in VARS \bullet e(i) = e_0(i))$

$\quad \wedge (\forall d \in ADR \bullet valid(d) = v_0(d)) \wedge e(a) \neq e(b) \wedge valid(e(b))\}$

`CALLEE_DEALLOC(a, b, tmp)`

$\{a \not\equiv tmp \wedge INV$

$\quad \wedge (\forall i \in VARS \bullet i \not\equiv a \wedge i \not\equiv a_{dealloc} \wedge i \not\equiv tmp \wedge i \not\equiv tmp_{dealloc} \implies e(i) = e_0(i))$

$\quad \wedge (\forall d \in ADR \bullet d \neq e_0(a) \wedge d \neq e(a) \wedge d \neq e(tmp) \implies valid(d) = v_0(d)))$

$\quad \wedge valid(e(b)) \wedge valid(e(a)) \wedge e(a_{dealloc})\}$

We construct the proof tableau of `CALLEE_DEALLOC(a, b)` macro as follows.

1 $\{a \not\equiv tmp \wedge INV e(a) \neq e(b) \wedge valid(e(b)) \wedge$
2 $\quad (\forall i \in VARS \bullet e(i) = e_0(i)) \wedge (\forall d \in ADR \bullet valid(d) = v_0(d))\}$①
3 `PRE_DEALLOC(a);`
4 $\{a \not\equiv tmp \wedge INV \wedge (\forall i \in VARS \bullet i \not\equiv a \wedge i \not\equiv a_{dealloc} \implies e(i) = e_0(i))$
5 $\quad \wedge (\forall d \in ADR \bullet d \neq e_0(a) \implies valid(d) = v_0(d)) \wedge$
6 $\quad e_0(a) \neq e(b) \wedge valid(e(b))\}$②
7
8
9 $\{a \not\equiv tmp \wedge INV \wedge valid(e(b)) \wedge (\forall i \in VARS \bullet e(i) = e_1(i)) \wedge$
10 $\quad (\forall d \in ADR \bullet valid(d) = v_1(d))\}$③
11 `tmp := copy(b);`
12 `tmp_dealloc:=true;`
13 $\{a \not\equiv tmp \wedge INV \wedge valid(e(b)) \wedge valid(e(tmp)) \wedge fresh(tmp) \wedge e(tmp_{dealloc}) \wedge$
14 $\quad (\forall i \in VARS \bullet i \not\equiv tmp \wedge i \not\equiv tmp_{dealloc} \implies e(i) = e_1(i)) \wedge$
15 $\quad (\forall d \in ADR \bullet d \neq e(tmp) \implies valid(d) = v_1(d))\}$④
16
17
18 $\{a \not\equiv tmp \wedge INV \wedge valid(e(tmp)) \wedge (\forall i \in VARS \bullet e(i) = e_2(i)) \wedge$
19 $\quad (\forall d \in ADR \bullet valid(d) = v_2(d))\}$⑤
20 `a := func(tmp, true);`
21 `tmp_dealloc = false;`
22 `a_dealloc := true;`
23 $\{a \not\equiv tmp \wedge fresh(a) \wedge \neg valid(e(tmp)) \wedge valid(e(a)) \wedge \neg e(tmp_{dealloc}) \wedge$
24 $\quad (\forall i \in VARS \bullet i \not\equiv a \wedge i \not\equiv a_{dealloc} \wedge i \not\equiv tmp_{dealloc} \implies e(i) = e_2(i))$⑥a $\wedge$
25 $\quad (\forall d \in ADR \bullet d \neq e(a) \implies valid(d) = v_2(d))$⑥b $\wedge valid(e(b)) \wedge e(a_{dealloc})\}$⑥
26
27
28 $\{a \not\equiv tmp \wedge INV$⑦a $\wedge valid(e(a)) \wedge valid(e(b)) \wedge e(a_{dealloc}) \wedge$
29 $(\forall i \in VARS \bullet i \not\equiv a \wedge i \not\equiv a_{dealloc} \wedge i \not\equiv tmp \wedge i \not\equiv tmp_{dealloc}$
30 $\quad \implies e(i) = e_0(i))$⑦b
31 $\quad \wedge (\forall d \in ADR \bullet d \neq e_0(a) \wedge d \neq e(a) \wedge d \neq e(tmp) \implies valid(d) = v_0(d))$⑦c$\}$

Listing 6.8: Tableau of `CALLEE_DEALLOC(a, b)` macro

Assumption ① Show $e(a) \neq e(b)$ and $valid(e(b))$ are assumed to be true in the entry condition of the macro.

Reasoning about ② Show $e(b) \neq e_0(a)$ and $valid(e(b))$ are both true in the post condition of pre-deallocation macro (refer to Theorem 6.2).

Assumption ③ Define $e_1(i)$ and $v_1(d)$ to store the values of variables and validity of addresses respectively after `PRE_DEALLOC` macro.

Reasoning about ④ Show $valid(e(b))$, $valid(e(tmp))$, $fresh(tmp)$ and $e(a_{dealloc})$ are true in the post-condition.

From line 11, we get $e(tmp_{dealloc})$.

$valid(e(tmp)) \wedge fresh(tmp)$ is included because we make a fresh copy of variable b to tmp.

$valide(e(b))$ remains true in the post-state because our macro in line 10 and 11 does not de-allocate anything.

Assumption ⑤ Define $e_2(i)$ to store the values of variable i at ⑤. Also, we define $v_2(d)$ to store the validity of address d.

Assumption ⑥ Show $fresh(a) \wedge \neg valid(e(tmp)) \wedge valid(e(a))$ holds true from the function behaviour (see Assumption 11). And $\neg e(tmp_{dealloc}) \wedge e(a_{dealloc})$ is included from line 19 and 20.

Reasoning about ⑦ₐ Show that INV holds at the end of macro.

$$INV : \forall i, j \in VARS \bullet inv\_dealloc(i, j)\text{Ⓐ} \wedge \forall i \in ARRVARS \bullet inv\_arr(i)\text{Ⓑ}$$

Proof. [Reasoning Deallocation Invariant Ⓐ]
Let $i, j \in VARS$ be the witness variables. We must show $\forall i, j \bullet inv\_dealloc(i, j)$: $e(i_{dealloc}) \wedge e(j_{dealloc}) \wedge i \not\equiv j \implies e(i) \neq e(j)$ holds true at ⑦ₐ.

As $inv\_dealloc(i, j)$ is symmetric, we can swap variable i and j without breaking the invariant, so $inv\_dealloc(i, j) \iff inv\_dealloc(j, i)$ and the reasoning just needs to consider three cases:

- Case 1: i, j includes a

  Given $i \equiv a$ (or equivalently $j \equiv a$), the invariant can be rewritten as:

  $$inv\_dealloc(a, j) : (e(a_{dealloc}) \wedge e(j_{dealloc}) \wedge a \not\equiv j) \implies e(a) \neq e(j)$$

  Assume that all the preconditions in $inv\_dealloc(a, j)$ are true, including $j \not\equiv a$. By $fresh(a) : j \not\equiv a \implies e(j) \neq (a)$ from ⑥, we have $e(j) \neq e(a)$ and conclude $inv\_dealloc(a, j)$ is true in the post condition.

- Case 2: i, j includes tmp

  Given $i \equiv tmp$ (or equivalently $j \equiv tmp$), the invariant can be rewritten as:

  $$inv\_dealloc(tmp, j) : (e(tmp_{dealloc}) \wedge e(j_{dealloc}) \wedge tmp \not\equiv j) \implies e(tmp) \neq e(j)$$

  We know $e(tmp_{dealloc}) = false$ from ⑥, and therefore $inv\_dealloc(tmp, j)$ is true after the macro.

- Case 3: i, j does NOT include a or tmp

  Let i, j be variables such that $i \not\equiv a$ and $i \not\equiv tmp$ (and $j \not\equiv tmp$ and $j \not\equiv tmp$).

  The instructions of callee macro do not change anything, except for a and tmp. That means, $e(i_{dealloc}) = e_2(i_{dealloc})$ and $e(i) = e_2(i)$ for $i \not\equiv a$ and $i \not\equiv tmp$. Since $inv\_dealloc(i, j)$ was true in ⑤, we therefore can conclude $inv\_dealloc(i, j)$ is still true in the post condition.

□

Proof. [Reasoning Array Invariant Ⓑ]

We must show $\forall i \in ARRVARS \bullet e(i_{dealloc}) \implies valid(e(i))$ holds true in the post-state.

Let $i \in ARRVARS$ such that $e(i_{dealloc})$ is true in the post-state, and we have to show $valid(e(i))$.

- Case 1: $i \equiv a$

  $inv\_arr(a) : e(a_{dealloc}) \implies valid(e(a))$ holds true because we have $valid(e(a))$ in ⑥.

- Case 2: $i \equiv tmp$

  $inv\_arr(tmp) : e(tmp_{dealloc}) \implies valid(e(tmp))$ is true since we have $\neg e(tmp_{dealloc})$ in ⑥.

- Case 3: $i \not\equiv a$ and $i \not\equiv tmp$

  We know $e(i) \neq e(a)$ because of $fresh(a)$.

  From $inv\_arr(i)$ at ⑤, we have $e_1(i_{dealloc}) \implies v_1(e_1(i))$

  From ⑥a, we have $e(i) = e_2(i)$ because $i \not\equiv a \wedge i \not\equiv a_{dealloc} \wedge i \not\equiv tmp_{dealloc}$.

  From ⑥b, we have $valid(e(i)) = v_2(e(i))$ because $e(i) \neq e(a)$ and $e(i_{dealloc}) = e_2(i_{dealloc})$

  Therefore, the validity must remain unchanged as ⑤.

$$valid(e(i)) = v_2(e(i)) = v_2(e_2(i))$$

□

Reasoning about ⑦b Show $\forall i \in VARS \bullet i \not\equiv a \wedge i \not\equiv a_{dealloc} \wedge i \not\equiv tmp \wedge i \not\equiv tmp_{dealloc} \implies e(i) = e_0(i)$ is true in the post condition.

Proof. Let $i \in VARS$ be a variable such that $i \not\equiv a$, $i \not\equiv a_{dealloc}$, $i \not\equiv tmp$ and $i \not\equiv tmp_{dealloc}$. We must show $e(i) = e_0(i)$.

From ⑥a because $i \not\equiv a \wedge i \not\equiv tmp_{dealloc} \wedge i \not\equiv a_{dealloc}$, we have $e(i) = e_2(i)$...(i)

From ④ because $i \not\equiv tmp \wedge i \not\equiv tmp_{dealloc}$, we have $e(i) = e_1(i)$...(ii)

From ③ and ②, because $i \not\equiv a$ and $i \not\equiv a_{dealloc}$, we get $e_1(i) = e_0(i)$...(iii)

By combining (i) (ii) (iii), with the predicate $i \not\equiv a$, $i \not\equiv a_{dealloc}$, $i \not\equiv tmp$ and $i \not\equiv tmp_{dealloc}$ we can therefore conclude

$$e(i) = e_2(i) = e_1(i) = e_0(i)$$

□

Reasoning about ⑦c Show $\forall d \in ADR \bullet d \neq e_0(a) \wedge d \neq e(a) \wedge d \neq e(tmp) \implies valid(d) = v_0(d)$ is true in the post condition.

Proof. Let $d \in ADR$ be an address such that $d \neq e_0(a)$ and $d \neq e(a)$ and $d \neq e(tmp)$. We must show $valid(d) = v_0(d)$.

From ⑥a because $d \neq e(a)$, we have $valid(d) = v_2(d)$...(i).

From ⑥b $\forall i \not\equiv a \implies e(i) = e_2(i)$. As $tmp \not\equiv a$, we have $e(tmp) = e_2(tmp)$.

From ④ $e(tmp) = e_2(tmp)$ and from ⑤, we get $d \neq e_2(tmp) \implies v_2(d) = v_1(d)$...(ii)

From ③ and ②, we get $v_1(d) = v_0(d)$ because $d \neq e_0(a)$...(iii).

By combining (i), (ii) and (iii), because $d \neq e(a) \wedge d \neq e(tmp) \wedge d \neq e_0(a)$, we can therefore conclude

$$valid(d) = v_2(d) = v_1(d) = v_0(d)$$

□

6.4 Automatic Proofs by Boogie

In the previous section, we have formally defined the deallocation invariant along with theorems of 8 deallocation macros. These properties are verified by hand to prove our invariant is preserved by each of our macros so that no double freeing problems would occur in our generated code.

In this section, we carry out the proofs of our invariant and macros by using the automatic theorem prover Boogie (Leino, 2008) which generates verification conditions from Boogie programs, and passes them to the SMT solver Z3 to verify the program properties. Boogie project is being developed by Microsoft Research, but is open-source (https://github.com/boogie-org/boogie).

We have mapped our invariant and macros to a Boogie program, as shown in Appendix A. There are two steps to this mapping: declarations and macro construction.

6.4.1 Declaration

```
0  // User−defined Type declaration
1  type VAR; // Generic variable types
2  type AVAR; // Array variable
3  type ADDR; // Adrress variable
4  // Map types
5  var e: [AVAR]ADDR;// Map an array variable to its addresses.
6  var dealloc: [AVAR]bool;// Indicate the deallocation flag for a array variable
7  // Indicate an address is valid if it has been heap−allocated, and not yet freed.
8  var valid: [ADDR]bool;
9  // define INV to describe deallocation invariant: inv_dealloc(i, j), inv_arr(i)
10 function INV(e: [AVAR]ADDR, dealloc: [AVAR]bool, valid: [ADDR]bool)
11 returns (r: bool);
12 axiom
13 (
14   ∀ e: [AVAR]ADDR, dealloc: [AVAR]bool, valid: [ADDR]bool •
15     INV(e, dealloc, valid)
16     ⟺ (∀ i,j: AVAR • dealloc[i] ∧ dealloc[j] ∧ i ≠ j
17                 ⟹ e[i] ≠ e[j]) // inv_dealloc (i, j)
18         ∧ (∀ i: AVAR • dealloc[i] ⟹ valid[e[i]]) // inv_arr(i)
19 );
```

Listing 6.9: Type declarations and Invariant

The declaration consists of types and invariant. We first need to declare types and invariant used in our macros, as shown in Listing 6.9, and also need *map* types to map one type to another, e.g. e maps an array variable to its memory address, and *dealloc* maps an array variable to the value of its deallocation flag (a boolean value). And *valid* maps an address to its boolean validity.

Second, we declare our deallocation invariant as a *function INV*. And then we postulate the properties of function *INV* by using an *axiom* for verifying our invariant is preserved by pre- and post-states of each macro. Our axiom combines single deallocation flag (see $inv\_dealloc(i, j)$ in Definition 6.5) and valid address invariant (see $inv\_arr(i)$ in Definition 6.5) to ensure only one variable with true flag value allows freeing the heap-allocated memory space, and guarantee that memory address is valid, which has not been freed yet.

So in our example axiom, variable i is an array type *AVAR*, and then the map selection `dealloc[i]` denotes variable i's deallocation flag value (or $e(i_{dealloc})$). Likewise, `e[i]` denotes the memory address that array variable i points to, or equivalently $e(i)$. And `valid[e[i]]` indicates if the address of array variable i is valid (or $valid(e(i))$).

6.4.2 Macro Construction

We define each macro as a *procedure* along with an *implementation*. A procedure includes the macro name, input parameters of the macro and a set of execution tracks, specified by pre and post-conditions with a combination of *requires*, *modifies* and *ensures* clauses. The implementation contains the actual code of the macro.

Example 6.10 *Consider caller macro (see Theorem 6.8) as an example. The Hoare triple of caller macro is listed:*

$\{a \not\equiv tmp \wedge INV \wedge (\forall i \in VARS \bullet e(i) = e_0(i)) \wedge$

$(\forall d \in ADR \bullet valid(d) = v_0(d)) \wedge e(a) \neq e(b) \wedge valid(e(b))\}$

CALLER_DEALLOC(a, b)

$\{a \not\equiv tmp \wedge INV \wedge valid(e(a)) \wedge e(a_{dealloc}) \wedge$

$(\forall i \in VARS \bullet i \not\equiv a \wedge i \not\equiv a_{dealloc} \wedge i \not\equiv tmp \wedge i \not\equiv tmp_{dealloc} \implies e(i) = e_0(i)) \wedge$

$(\forall d \in ADR \bullet d \neq e_0(a) \wedge d \neq e(a) \wedge d \neq e(tmp) \implies valid(d) = v_0(d))\}$

```
0  procedure caller_dealloc(a: AVAR, b: AVAR, tmp: AVAR) returns();
1    requires tmp ≠ a;
2    requires INV(e, dealloc, valid) ∧  e[a] ≠ e[b] ∧ valid[e[b]];
3    modifies e, dealloc, valid;
4    ensures (∀ i: AVAR • i ≠ a ∧ i ≠ tmp ⟹ e[i] = old(e[i]));
5    ensures (∀ d: ADDR • d ≠ old(e[a]) ∧ d ≠ e[a] ∧ d ≠ e[tmp]
6                ⟹ valid[d] = old(valid[d]));// Address validity
7    ensures (∀ i: AVAR • i ≠ a ∧ i ≠ tmp
8                ⟹ dealloc[i] = old(dealloc[i]));// Deallocation flag
9    ensures valid[e[a]];
10   ensures dealloc[a];
11   ensures INV(e, dealloc, valid);
12 implementation caller_dealloc(a: AVAR, b: AVAR, tmp: AVAR)
13 returns ()
14 {
15   var ret:  ADDR;// Local variable 'ret' stores the address
16   call pre_dealloc(a);
17   call ret := copy(b);
18   e := e[tmp := ret]; // e[tmp]:= ret
19   dealloc := dealloc[tmp := false]; // dealloc[tmp] := false
20   call ret := reset_caller_func(tmp, false); // ret := func(b, false);
21   e := e[a := ret]; // e[a]:=ret;
22   if(e[a] ≠ e[tmp]){
23     call freed(tmp); // free variable 'tmp'
24   }
25   dealloc := dealloc[a := true]; //dealloc[a] := true
26 }
```

Listing 6.10: Caller Macro in Boogie

We will transform caller macro to a procedure implementation in Boogie (as shown in Listing 6.10). The program is explained as follows.

Procedure We define the macro as a procedure which takes array variables a and b as input parameters, and *tmp* as block-scoped array variable.

We express the pre-conditions of caller macro as *requires* clause that our invariant holds before the macro, and also use one `modifies` clause to specify the variables that will be changed in the implementation of our macro. We do not include $e(i) = e_0(i)$ and $valid(d) = v_0(d)$ in the pre-conditions of `caller_dealloc` macro because procedure implementation is two-state contexts in Boogie, and `old` expression is provided to access the value on entry to the procedure. So `old(e[i])` denotes $e_0(i)$, and `old(valid[d])` refers to $v_0(d)$.

We encode post-conditions as a number of *ensures* clauses, such as our invariant, address validity, etc. For example,

```
ensures (∀ i: AVAR • i ≠ a ∧ i ≠ tmp ⟹ e[i] = old(e[i]));
```

the above post-condition (ensure clause) specifies the final values of all the other variables, except for i and *tmp*, are the same as their initial values. And we also include another post-condition:

```
ensures (∀ i: AVAR• i≠a ∧ i≠tmp ⟹ dealloc[i]=old(dealloc[i]));
```

and verify that, apart from variables a and *tmp*, the deallocation flag values of all the other variables remain unchanged on exit of procedure *caller_macro*, so are the same as their values on entry.

```
0  implementation caller_dealloc(a: AVAR, b: AVAR, tmp: AVAR) returns ()
1  {
2    var ret: ADDR;// Local variable 'ret' stores the address of a function call
3    call pre_dealloc(a); // PRE_DEALLOC(a);
4    call ret := copy(b); // ret = copy(a);
5    e := e[tmp := ret]; // e[tmp] = ret;
6    dealloc := dealloc[tmp := false]; // tmp_dealloc = false;
7    call ret := func(tmp, false);// ret = func(tmp, false);
8    e := e[a := ret]; // a =ret;
9    if(e[a] ≠ e[tmp]){call freed(tmp);}// if (a ≠ tmp)  {free(tmp);}
10   dealloc := dealloc[a := true]; //a_dealloc = true;
11 }
```

Listing 6.11: Caller Macro Implementation in Boogie (Comment: C code)

Implementation Convert the actual code of our macro into the below Boogie implementation (see Listing 6.11).

We introduce local variable *ret* to temporarily store the memory space returned by procedure *copy* at line 5 or *func* at line 8 in Boogie program.

The call statement `call pre_dealloc(a)` invokes procedure *pre_dealloc* which checks flag value of variable *a* to free the memory address of variable *a*. If procedure *pre_dealloc* is called while satisfying all its preconditions, then Boogie assumes the post-conditions of procedure *pre_dealloc* to be true when the call finishes. As such, the specifications (pre- and post-conditions) are mandatory to define the behaviour of a procedure whereas the implementation can be optional. For example, procedure *freed* can be written:

```
0 procedure freed(a: AVAR) returns ();// 'freed' procedure
1     requires valid[e[a]]; // A valid address of 'a' on entry
2     modifies valid;
3     ensures valid[e[a]] = false; // An invalid address of 'a' on exit
4     // Other addresses remain the same validity upon procedure return
5     ensures (∀ d: ADDR • d ≠ e[a] ⟹ valid[d] = old(valid[d]));
```

Listing 6.12: Procedure *freed* in Boogie

This procedure does not have any implementation but includes a list of specifications to specify that the address of variable *a* on entry is valid and has not been freed yet, and invalidate the address of variable *a* on exit. Note that the complete code of procedures *pre_dealloc*, *copy* and *reset_caller_func* is shown in the Boogie program A.1.

6.4.3 Proof Results

Boogie verifier automatically transforms our Boogie program into a set of verification conditions and validates these pre- and post-conditions with a theorem prover (Z3) to prove the correctness of given program. Boogie verifier can provide counter examples to explain the potential errors in the program if the proof fails.

Example 6.11 *Consider our caller macro again. We delete the pre-condition* $a \not\equiv tmp$ *at line 2 in Listing 6.10 and enable* `captureState` *feature to capture*

intermediate states of each statement in the implementation body so that we can keep track of the value change of each variable in the program. Then we try to verify the program with Boogie again. And the proof fails because the post-condition at line 6 may not hold at the end of procedure. And we obtain a counter example in the following trace.

Table 6.2: Counter Example from Boogie Verifier

Line No. Statement	Variable Address				Address Validity		
	$e(a)$	$e(b)$	$e(tmp)$	ret	valid('5)	valid(e(b))	valid(ret)
16. `var ret:ADDR`	'19	'15	'19	'13	False		
17. `call pre_dealloc(a)`		'15		'13	False	True	
18. `call ret := copy(b)`		'15		'5	True	True	True
19. `e[tmp]:=ret`	'5	'15	'5	'5	True	True	True
20. `dealloc[tmp]:=false`	'5	'15	'5	'5	True	True	True
21. `ret := func(tmp, false)`	'5	'15	'5	'7	True	True	True
22. `e[a]:= ret`	'7	'15	'7	'7	True	True	True
27. end	'7	'15	'7	'7	True	True	True

The counter example in Table 6.2 shows the memory addresses of variables at each intermediate step and their corresponding validity. From the table, we know variable a and tmp are the same array variable as they are update simultaneously at line 19 and 22, and the assumption $a \not\equiv tmp$ is removed from the Boogie program.

The address $'5$ is the return value of procedure *copy* at line 18 and changes its validity from *false* to *true*. Because of variable update at line 19, we have a shared address for variables a, tmp and ret ($e(a) = e(tmp) = ret =' 5$).

Likewise, the address $'7$ is the return value of procedure *func* at line 21 and the variable update at line 22 makes variables a, tmp and ret point to the new address ($e(a) = e(tmp) = ret =' 7$)

At the end of the procedure (line 27), the address $'5$ has changed its validity during the macro and does not belong to any variable (a, tmp or $old(a)$) in the predicate of address validity post-condition:

```
ensures (∀ d: ADDR • d ≠ old(e[a]) ∧ d ≠ e[a] ∧ d ≠ e[tmp]
            ⟹ valid[d] = old(valid[d]));// Address validity
```

So Boogie indicates that the post-condition of unchanged address validity may not hold on exit of caller macro procedure. This failure happens because variables a and tmp are the same variables ($a \equiv tmp$), and we conclude that $a \not\equiv tmp$ is necessary for proving the correctness of caller macro program.

The Boogie program of all our macros using this pre-condition $a \not\equiv tmp$ is shown in Listing A.1 of Appendix A, and the program has been verified to be **valid without any error** by Boogie verifier (version 2.3.0.61016).

Such a pre-condition $a \not\equiv tmp$ was originally omitted at our specifications. But Boogie helps us discover the potential mistakes of our by-hand proofs, and lets us strengthen both of our manual and automatic verification. So Boogie can be a complementary tool to our manual proofs as it automatically double-checks the correctness of our proofs so that mistakes during the proving process can be avoided. But good and sound manual program reasoning skill is still required to use the automatic Boogie verification. For example, we need to correctly transform the specifications and verification conditions into a Boogie program, and also need to be able to interpret the complicated and lengthy counter example from Boogie verifier.

Chapter 7

Code Generator

This chapter details the code generator and code optimisation using analysis results. Firstly, the Whiley compiler reads and translates a Whiley program into Whiley Intermediate Language (WyIL) code. Secondly, our code generator works with copy and deallocation analysers to produce efficient C code.

The copy analyser detects and removes unnecessary copies at generated code so that we can greatly decrease expensive overheads caused by array copies and thus improve the speed-up of program execution. Also, our deallocation analyser helps choose suitable de-allocation macros and apply them to generated code. So the optimised code with our macros can keep single ownership of the shared memory and therefore we can safely over-write unused array pointers and ensure no delete occurs on the same memory space twice.

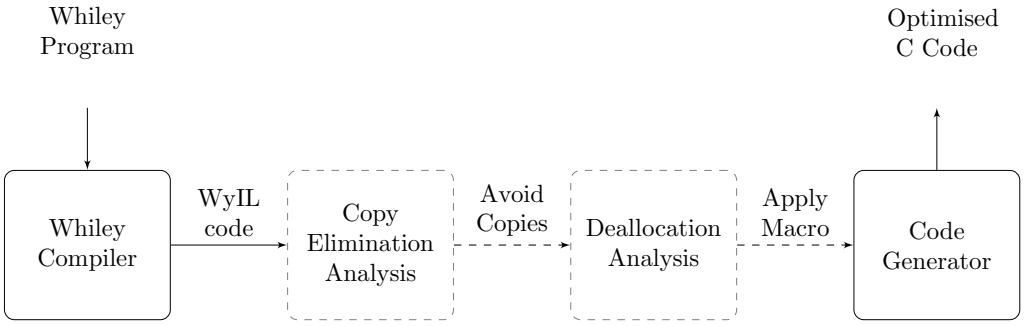


Figure 7.1: Flow chart of code generation and optimisation (dashed box)

7.1 Naive Code Generator

Procedure 7.1 Generating naive C code
Input: The input *WyIL* code, produced by Whiley Compiler
Output: The *list* of generated C code

```
1: list := []
2: for each func function in WyIL code do
3:     list.append(Define_Function(func))
4:     vars = variable tables of func
5:     for each var in vars do
6:         list.append(Declare_Variable(var, func))
7:     end for
8:     for each code in function body do
9:         list.append(Generate_Code(code, func))
10:    end for
11: end for
12: return list
```

The code generator without any code optimisation will translate WyIL code into naive/unoptimised C code, semantically equivalent to original program, which makes copies before any change to avoid side effects and also lacks memory deallocation. A typical function includes three parts:

- **Function signature** consists of return type, function name and parameters.
- **Variable declaration** defines the type of a variable and its initial value.
- **Function body** contains a collection of statements, which is translated from each line of WyIL code.

Each part will be discussed in following sections.

7.1.1 Function Signature

The code generator extracts function name, parameters and parameter types and return type to produce the function signature. For each array parameter, an additional size variable is then appended to the function signature, to pass the size of input array from caller site to called function. If the return is an array, we also pass the size variable as call-by-reference parameter to the called

function, so that the size of output array can be changed by called function and the updated size value is visible at caller site.

```
// private function func(int[] a) -> int[] r:
// 'a_size': size of input array 'a'
// 'ret size': Cally-by-reference size of output array 'r'
int64_t* func(int64_t* a, size_t a_size, size_t* r_size){
  int64_t* r;
  ....

  // Update return array size
  *r_size = 10;
  return r;
}
```

For example, input and output of function *func* are array a and r respectively. We then pass the size variable of input array *a_size* as one parameter to the called function, and also include the size variable of output array *ret_size* as call-by-reference parameter, as shown in above listing.

7.1.2 Variable Declaration

The code generator goes through each variable' names and type in a function to produces a list of variable declarations before function body. WyIL byte-code is register-based (Pearce and Groves, 2015a), and thus each parameter or variable is prefixed with %. Additionally, because of single assignment form (SSA) used in WyIL code, each variable is assigned once and must be defined before its use. We therefore have more number of registers at WyIL level than at source Whiley level. Consider the below Whiley program.

```
function func(int[] x, int num) -> int[]:
  x[0] = num
  return x
```

The converted WyIL code is as below:

```
private function func(int[], int) -> (int[]):
// %0: x and %1: num
body:
  const %3 = 0 : int // %3: constant value 0
  update %0[%3] = %1 : int[] -> int[] // x[0] = num
  return %0 // x
```

Registers *%0* and *%1* hold the values of first x and second parameters *num* respectively on the function entry. Register *%3* loads constant value 0 from `const` WyIL code to access array item at index of 0.

By default, an integer variable is declared as signed 64-bit type (`int64_t`) to have the maximal and minimal range in 64-bit operation system. For an array typed variable, we declare it as the below pointer and also include a size variable to store its value.

```
1 int64_t* a = NULL; // int[] a
2 size_t a_size = 0; // size variable
```

The reasons that we choose heap-allocated pointers over stack arrays are:

- Heap pointer can make use of all available physical memory that operation system provides at most, so give bigger array capacity to run on large-scaled benchmarks, whereas stack arrays have smaller limitation on array size;
- Heap pointer can be resized at run-time whereas static array size is determined at declaration and can not be altered once compiled.

If the value range of an integer or array can be statically known and estimated by our bound analysis, the code generator can use more fitting integer types to store the values.

7.1.3 Function Body

The code generator maps each WyIL code of function body to its type and then translates it into a sequence of C code. The following shows a list of crucial code types for generating and optimising code.

- `code == arraygenerator( a = (value, size) )` An array generator statement creates an array variable *a* of given *size* and initialises each array element with given *value*.

  For a one-dimensional array, we assume the array stores signed 64-bit integers as default type. And we define a single dimensional array as a pointer with an extra size variable to keep track of its array length using above `NEW_1DARRAY` macro. We also includes a check after memory allocation to ensure the array pointer points to a valid memory address.

```
// Create an array of provided type and size and fill with given value
#define NEW_1DARRAY(a, value, size, type)
({
  a_size = size;
  a = (type*)malloc(a_size*sizeof(type));
  if(a == NULL){
    fputs("fail to allocate the memory at _NEW_1DARRAY\n",
    stderr);
    exit(-2);
  }
  // Initialize each item value of array 'a'
  for(size_t i=0;i<a_size;i++){
    a[i] = value;
  }
})
```

For a two-dimensional array, we first map it to 1D array and specify its size variable to the total number of array items, i.e. $width \times height$, and then populate the array's value. Therefore, we access the array item at i row and j column by using $a[i * width + j]$, instead of $a[i][j]$.

In doing so, all array elements are allocated on contiguous memory space so that the data locality can be improved. Since each sub-array has the same length, the dynamical-sized array is not supported in our project.

- `code == assign( a = b )` An assignment statement assigns value b to variable a. For an integer-typed assignment, we do not need to make a copy as primitive integers are declared in stack and automatically copied before any change occurs.

  For an array assignment $a = b$ our naive code without optimisation always makes a copy of right-handed side variable b and assigns the copied one to left-handed side a. In addition, the old array size is propagated to the new array.

```
// Make a copy of array 'b'
#define COPY(a, b, type)
({
  a_size = b_size;
  a = (type*) malloc(a_size * sizeof(type));
  if (b == NULL) {
    fputs("fail to malloc at COPY macro\n", stderr);
    exit(-2);
  }
  memcpy(a, b, b_size * sizeof(type));
})
```

Making a copy of right-handed side variable in each assignment slows down program execution. Thus, we use copy elimination and de-allocation analysers (see Section 7.2) to find out and remove extra copies from some assignments and improve the efficiency.

- `code == binOp( a = (b, c) )` A binary operator manipulates variable b and c with operator *binOp*, and stores the result to variable a.

```
a = b + c; // add a = (b, c)
a = b * c; // mul a = (b, c)
```

The common operators include `add`ition (+), `sub`traction (-), `mul`tiplication (*), `div`ision (/) and `rem`ainder (%), etc.

```
// Detect the addition overflow 'a = b + c'
#define INT_ADD_OVERFLOW(a, b, c)
({
  if(__builtin_add_overflow(b, c, &a)){
    fputs("Detected an add overflow \n", stderr);
    exit(-2);
  }
})
```

We may encounter arithmetic overflows for unbounded integers, and thus use GCC built-in functions (Stallman and the GCC Developer Community, 2003) to check whether the operation causes overflow or not, and throw out run-time exceptions if detected. By default, the overflow check is disabled because we declare all integer variables as signed 64-bit integers, and its range ($-2^{63}+1 \sim 2^{63}-1$) is large enough to perform all normal arithmetic operations on a 64-bit operation system.

- `code == label( blklab )` A label statement specifies the block label, which is composed of an identifier and block number (e.g.`blklab1`), to indicate the location of block within source code.

- `code == if( OP(a, b) goto blklab )` An IF statement compares the values of variable a and b with operator OP, and then specifies the block label *blklab* that is to be executed when the condition is met (true). Common comparing operators OP include *eq* (==), *neq* (!=), *lt* (<), *le* (<=), *gt* (>) and *ge* (>=).

```
1 // if( ge(x, 10) goto blklab1 )
2 if(x>=10){goto blklab1;}
3 ...
4 blklab1:; // Block label that 'goto' branches to
```

- `code == loop([a, ...], [codes])` A loop repeatedly executes a list of *codes* until any loop condition, comparing the value of a loop variable a, is no longer true. We use a while loop along with one or a series of conditional checks, to decide whether to continue or terminate the loop, as shown in below:

```
1 // loop ([i, 10, sum], [ sum = sum + i, i = i + 1 ])
2 while(true){
3   if(i > 10){goto blklab1;}// loop condition
4   sum = sum + i;
5   i = i + 1;
6 }
7 blklab1:; // Loop exit label
```

  A loop may contains one or more inner loops, and our code generator therefore goes into each inner loop recursively, and then put it within the outer loop to form a hierarchy of loop nests.

- `code == invoke( a = func(b, c, ...) )` A function call passes one or more parameters $b, c, \ldots$ to the called function *func*, and returns the result to variable a if return value is required.

  Our naive code always copies an array parameter first and then pass the copied one to called function, to ensure all changes to parameters made by the function call will not affect the original values at caller site. By doing so, our naive code conforms to immutable value semantics in functional programming language and thus does not cause any side effect.

```
1 // a = func(b)
2 // Make a copy at 'b' at function call 'func'
3 // Pass call-by-reference array size 'a_size' to 'func'
4 a = func(COPY(b), b_size, &a_size);
```

  However, the copying of array parameters increases the overheads when arrays are large and makes the execution slow. Also, the de-allocation

of array parameters is another performance issue because it may lead to memory leaks or worse double freeing problem.

Our copy elimination and deallocation analysers can work together to sort out the needs of parameter copies and determine their deallocation responsibility (see Section 7.2)

- `code == assert( expr )` An assert statement contains a block of byte-codes to evaluate an condition *expr*. If the assertion fails, an exception is thrown out to stop the program execution and ensure the safety.

```
1  // assert ( expr )
2  {// Beginning of assertion block
3    if(expr){goto blklab0;}// If expr is evaluated to true, go to blklab0
4    fprintf(stderr,"fail");// expr is evaluated to false, throw error
5    exit(-1);// Stop the program
6  blklab0:;
7  }// End of assertion block
```

- `code == return( a )` A return statement passes back variable *a* to the caller when the invoked function finishes. In the case that *a* is an array return, as its array size is stored separately, the size variable *a_size* can not be passed back to caller site at the same time as return array variable *a* because C language restricts a single return. To address this issue, we use below workaround to handle an array return.

```
1  // 'a' is an array returned by function 'func'
2  // 'a_size' is updated by 'func' function and the change is visible at
       method 'main
3  int64_t* func(int64_t* b, size_t b_size, size_t* a_size){
4    ...
5    *a_size = 10; // Update the size of array 'a'
6    return a; // Return array
7  }
8  // Method 'main'
9  void main(){
10   int64_t* a;
11   size_t a_size = 0;
12   int64_t* b;
13   size_t b_size;
14   ...
15   // Pass 'a_size' as call-by-reference parameter
16   a = func(b, b_size, &a_size);
17   assert(a_size == 10);
18 }
```

The size variable *a_size* is passed as a call-by-reference parameter to called function *func*, so that its value is updated before the return. After

the function call, we will have both output array and size updated by called function *func*, and those changes are visible at caller site.

```
1  // Function 'func' may change 'b' array and may return 'b' array
2  // If not, return new array 'c'
3  function func(int[] b, int num) -> int[]:
4      int[] c = [0;3] // c[0] = 0
5      if num > 10:
6          b[0] = num
7          return b
8      else:
9          return c
10 // Method 'main'
11 method main(System.Console sys):
12     int[] b = [2;3] // b[0] = 2
13     int[] tmp = func(b, 11) // function call
14     b = tmp // b[0] = tmp[0] = 11
15     assert b[0] == 11
16     sys.out.println(b[0])
17     b = func(b, 65536)  // function call
18     sys.out.println(b[0])
19     assert b[0] == 65536
```

Listing 7.1: Example Whiley program

```
1  // function func(int[] b, int num) -> int[]:
2  int64_t* func(int64_t* b, size_t b_size, int64_t num,
3    size_t* ret_size){
4    int64_t* _6 = NULL; size_t _6_size = 0;
5    int64_t*  c = NULL; size_t  a_size = 0;
6    //arraygen %6 = [0; 3] : int[]
7    NEW_1DARRAY(_6, 0, 3, int64_t); // _6_size = 3;
8    //assign c = %6  : int[]
9    c = COPY(_6, int64_t); // c_size = _6_size;
10   //ifle %1, 10 goto blklab0 : int
11   if(num<=10){goto blklab0;}
12   //update b[0] = num
13   b[0] = num;
14   //return b
15   *ret_size = b_size;
16   return b;
17 blklab0:;
18   //return c
19   *ret_size = c_size;
20   return c;
21 }
```

Listing 7.2: Naive C code of function *func* (comments: WyIL code)

Example 7.1 *We will illustrate the procedure of generating naive code from a WyIL file with an example program as shown in Listing 7.1. Function func takes array b and integer num as inputs, and checks num value to decide whether to return an array b with update, or a new array c.*

At method 'main' in line 13, we make a function call and assign return

value to array tmp, and then over-write array b with array tmp. In line 17, we make another function call to update array b with larger value.

Function *func* Has argument array *b* and its size *b_size* and integer *num*. Also, an extra call-by-reference size variable *ret_size* is passed as an argument to function *func* to keep track the actual size of return array. And we declare all the local variables as follows:

- All integer typed variables are signed 64-bit integers (`int64_t`);
- All integer array typed variables are signed 64-bit integer pointers (`int64_t*`);
- All array size variables are defined as size type (`size_t`) as it can represent the size of any object in a program;
- The argument of return array size is declared as size typed pointers (`size_t*`), instead of a value, so that function *func* has direct access to modify its value and make the updates visible to the caller.

Whiley intermediate code replaces each target of every assignment with a new variable since it follows static single assignment form (Pearce and Groves, 2015a). Thus, we have `arraygen` code in line 7 to store the newly created array to a temporary variable *_6*. Then we have an assignment in line 9 to write temporary array *_6* to target variable *a*. Due to value semantics for each assignment, we therefore *make an extra copy in line 9*.

The return of function *func* is based on the value of passed *num* to determine to pass back array *x* or *c*. And before each return statement, we update the passed call-by-reference size argument *ret_size* with specified size variable of return array.

Method main Creates a new array using `NEW_1DARRAY` macro and makes two function calls on *func* and assigns the return to variable *x*. Similar to Function *func*, we use the same rule to declare the types of all local variables. And in naive/unoptimised code all the copies are needed to ensure right-handed

side variable will not be changed by an assignment and passed parameters will not affect the values at caller site, and achieve side effect-free function calls as well as assignments.

```
1  int main(int argc, char** args){
2    int64_t* _5 = NULL;  size_t _5_size = 0;
3    int64_t*  b = NULL;  size_t  b_size = 0;
4    int64_t* _8 = NULL;  size_t _8_size = 0;
5    int64_t* tmp = NULL; size_t tmp_size = 0;
6    int64_t* _18 = NULL; size_t _18_size = 0;
7    //arraygen %5 = [2; 3] : int[]
8    NEW_1DARRAY(_5, 2, 3, int64_t); // _5_size = 3;
9    //assign b = %5  : int[]
10   b = COPY(_5, int64_t); // b_size = _5_size;
11   //invoke (%8) = (b, 11) func : function(int[],int)->(int[])
12   _8 = func(COPY(b, int64_t), b_size, 11, &_8_size);
13   //assign tmp = %8  : int[]
14   tmp = COPY(_8, int64_t); // tmp_size = _8_size;
15   //assign b = tmp  : int[]
16   b = COPY(tmp, int64_t); // b_size = tmp_size;
17   //assert b[0] == 11
18   ASSERT(b[0] == 11);
19   //sys.out.println(b[0])
20   printf("%"PRId64"\n", b[0]);
21   //invoke (%18) = (b, 65536) func : function(int[],int)->(int[])
22   _18 = func(COPY(b, int64_t), b_size, 65536, &_18_size);
23   //assign b = %18  : int[]
24   b = COPY(_18, int64_t); // b_size = _18_size;
25   //assert b[0] == 11
26   ASSERT(b[0] == 65536);
27   //sys.out.println(b[0])
28   printf("%"PRId64"\n", b[0]);
29   //return
30   exit(0);
31 }
```

Listing 7.3: Naive C code of method *main* (comments: WyIL code)

Listing 7.3 shows the naive code of *main* method. In the first function call (line 13), array variable b explicitly is copied and passed to function *func*. Primitive typed variables (e.g. *num* and *b_size*) do not need *COPY* macro but can be automatically copied to function *func* because C programming language applies call-by-value approach to those arguments by default. Then the function result is assigned to a new fresh variable *_8*, which does not appear before, due to static single-assignment (SSA) form at intermediate level. In line 25, we have another function call and thus make a copy of array b.

In line 10, 16, 18 and 28 we have an assignment that requires the copy of right-handed side variables, Therefore, *we have six copies in main method.*

7.2 Code Optimisation and Integer Type Choice

The naive C code requires further optimisation to improve program efficiency. Before generating the optimised code, we have *function analysers* to pre-process each function by scanning each line of code and collecting the sets of read-write variables, return variables and live variables, and then keep trace of all analysis results for copy and deallocation analyser, to determine the optimisation for each code and produce corresponding optimised C code.

The naive C code makes a copy as default action for each assignment and function call, because of value semantics, but results in expensive overheads of program execution. Our copy analyser aims to remove unneeded copies from generated code and still keep the program running without any side effect.

The naive or copy eliminated C code has memory leak problem as all arrays are allocated on heap memory and require manual deallocation. Our deallocation analyser aims to automatically choose proper deallocation macros for each code so that the unused variables can be freed at run-time. Also, our macro has been designed to have single deallocation ownership and thus ensure the same memory space is never freed twice.

The default integer type for all unoptimised and optimised code is signed 64-bit integer (`int64_t`). Our bound analyser performs static range analysis to estimate the domain of every integers, and varies the used fixed-size integer types wherever possible.

7.2.1 Copy Elimination

Copying takes place at an array assignment, or array typed parameter passed to a function call. For an assignment $a = b$ where a and b are arrays, copy analysis takes out the copy of array b if variable b is not live/used afterwards and simply aliases the left and right variables.

Procedure 7.2 Removal of Copies using Copy Elimination Analysis

Input: Variable *var* at *code* in function *func*
Output: Return *true* if the copy of variable *var* is removed.

```
1: Variables
2:     MutateAnalyser: Read/Write Analyser
3:     ReturnAnalyser: Return analyser
4:     LiveAnalyser: Live variable analyser
5:     isLive: Is var still used/live after code in func
6:     isMutated: Is var mutated by called function callee
7:     isReturned: Is var returned by called function callee
8: end Variables
9: procedure IsCopyRemoved(var, code, func)
10:    if code is Assignment then
11:        // Check if var is used after code in func
12:        isLive ← LiveAnalyser.isLive(var, code, func)
13:        return ¬isLive // Remove copy when var is NOT live
14:    else if code is Function Call then
15:        callee = get called function from code
16:        param = map var to formal parameter of function callee
17:        // Check if param is mutated by function callee
18:        isMutated ← MutateAnalyser.isMutated(param, callee)
19:        // Check if param is returned by function callee
20:        isReturned ← ReturnAnalyser.isReturn(param, callee)
21:        isLive ← LiveAnalyser.isLive(var, code, func)
22:        if ¬isLive OR (¬isMutated AND ¬isReturned) then
23:            return true // Eliminate the copy
24:        end if
25:        return false // Keep the copy
26:    else// No needs to optimise the code
27:    end if
28: end procedure
```

For a function call $a = func(b)$ where b is an array, the naive code generator goes through each parameter of function call and makes one copy for each array parameter. Our copy analysis removes the copy of array parameter b if

- variable b at caller site is not live/used after the function call. Since dead b has no uses afterwards, its copy is unnecessary, or

- parameter b is not changed nor returned by called function *func*. So parameter b is read-only and not aliased to the return at function *func*. Since parameter b does not change during function call, it does not cause any side effect and thus its copy can be safely eliminated.

7.2.2 Deallocation Macro

Heap-allocated arrays are the source of memory leaks in our naive or copy eliminated code and require extra de-allocation to free their previously allocated memory space. If failing to do so, the amount of memory leaked will be accumulated as the program is run for long, and eventually exhaust all system memory. We go through naive and copy eliminated code and find the following memory leaks:

- Memory leaks for left-handed side at an assignment or a function call: An assignment or a function call directly writes a new value to the left variable without deallocation. The old value of left variable is still allocated on heap, and thus results in memory leaks.

- Memory leaks for function parameter: Once a call-back is finished, if none of called and caller function tries to de-allocate the parameter, it causes memory leaks. But if both of called function and caller try to free the same and shared parameter, then it leads to double free memory errors as no space can be deleted twice.

- Memory leaks for local variables: Local variables are not freed after a function terminates.

Our deallocation analyser designs a macro system to handle the above memory problems and splits the de-allocation work into pre-deallocation and post-deallocation macros, which each chooses the macro according to our deallocation rule. The analyser firstly creates one boolean flag variable for every heap-allocated array variable and associates the flag value with its array's deallocation responsibility at run-time. When a new variable takes over an old array, our macro will change flag value of relevant variables to ensure that single owner is responsible for deallocation, and that every array variable with true flag points to a valid address.

Procedure 7.3 Pre-deallocation macro by deallocation analysis

Input: *code* in function *func*
Output: A list of pre-deallocation *macros* suggested for *code* in *func*

```
 1: procedure choosePreDeallocMacro(code, func)
 2:     macros := []
 3:     if code is Assignment OR code is Function call OR
 4:      code is Array Generator then
 5:         lhs = left-handed side of code
 6:         if lhs is Array then
 7:             macros.append(PRE_DEALLOC(lhs))
 8:         end if
 9:     else if code is Return then
10:         ret = return variable of code
11:         for each var variable in func, except for ret variable do
12:             if var is Array then
13:                 macros.append(PRE_DEALLOC(var))
14:             end if
15:         end for
16:     else// No needs to use macro
17:     end if
18:     return macros
19: end procedure
```

Secondly, our analyser targets on array generator, assignment, function call and return statements, and applies `PRE_DEALLOC` macros on dead variable before each statement, i.e. left operand at an assignment or array generator, and target variable at a function call. So we can safely empty the memory space of target variable to store new values.

For a return statement, we keep return variable unchanged but free all local variables and function parameters, depending on their associated deallocation flags as they are out of the function scope. Further, because copy analyser may make multiple variables aliased to the same memory space, the deallocation of a return require extra owner check, which can be resolved by using our pre-deallocation macro.

Our analyser goes through each local variable and function parameter, and generates a list of `PRE_DEALLOC` macros before the return statement to free all used memory space in a function. And because the invariant of single deallocation owner holds by our deallocation macros, our `PRE_DEALLOC` macro can free any memory space only once, and thus avoid the problem of double

deletes on the shared memory.

Thirdly, our deallocation analyser chooses the post-deallocation macros to change the values of deallocation flag in the post state.

- For an array generator, we use `NEWARR` macro to assign true flag to target variable
- For an assignment, we choose between `ADD` or `TRANSFER` macro, depending on the copies of right variable, which may be removed by our copy analyser.
- For a function call, we have four kinds of post-deallocation macros: `RETAIN`, `RESET`, `CALLER` and `CALLEE` macros. The choice depends on true liveness of actual parameter at caller site and the variable properties (mutation and return) of its corresponding formal parameter at called function (see deallocation rule 6.1).

7.2.3 Code Optimisation and Generation

Once copy and deallocation analysers finish the optimisation of all functions, our code generator goes through each function and produces optimised C code as output, as shown in below Algorithm 7.5. For a function *func*, the code generation phase consists of three parts: function signature, variable declaration and body. First, our code generator constructs the function definitions (return type, name and parameters). But for each array typed parameter, our deallocation optimisation appends one extra boolean flag variable, next to the parameter in declaration, to indicate if the variable has true flag to free the allocated memory space. Secondly, our code optimiser appends the run-time deallocation variable for each local array variable, and initialises the value to be false. Thirdly, we go through each line of code in function body, check the code type (`assignment`, `function call` and `return`) to call the corresponding code optimiser, and then produce optimised C code with help from copy and deallocation analysers. We will discuss each code optimisation as follows.

Procedure 7.4 Post-Deallocation Macro by Deallocation Analysis

Input: One line of *code* in function *func*
Output: A list of post-deallocation *macros* suggested for *code* in *func*

```
1: Variables
2:     MutateAnalyser: Read/Write Analyser
3:     ReturnAnalyser: Return analyser
4:     LiveAnalyser: Live variable analyser
5:     CopyAnalyser: Copy elimination analyser
6:     macros: A list of macros used in code
7:     aParam: Actual parameter used in function call code
8:     fParam: Formal parameter used in definition of called function
9: end Variables
10: procedure computePostDeallocMacro(code, func)
11:     macros = []// Store all macros used in code
12:     if code is ArrayGenerator then
13:         target = array variable of code
14:         macros.append(NEWARR_DEALLOC(target))
15:     else if code is Assignment then
16:         lhs = left-handed side of code
17:         rhs = right-handed side of code
18:         if lhs is an Array AND
19:            CopyAnalyer.isCopyRemoved(rhs, code, func) then
20:             macros.append(TRANSFER_DEALLOC(lhs, rhs))
21:         else// Copy of rhs is NOT removed at code
22:             macros.append(ADD_DEALLOC(lhs, rhs))
23:         end if
24:     else if code is Function call then
25:         ret = return variable of code
26:         callee = called function of code
27:         for each aParam in code do// Iterate each actual parameter aParam
28:             if aParam is Array then// Macro is applied on array type only
29:                 isLive ← LiveAnalyser.isLive(aParam, func)
30:                 fParam = map aParam at caller func to
                                formal parameter at called function callee
31:                 isMutated ← MutateAnalyser.isMutated(fParam, callee)
32:                 isReturned ← ReturnAnalyser.isReturn(fParam, callee)
33:                 switch isMutated − isReturned − isLive do
34:                     case F-F-T ∨ F-F-F ∨ T-F-F
35:                         macros.append(RETAIN_DEALLOC(ret, aParam))
36:                     case F-T-F ∨ T-T-F
37:                         macros.append(RESET_DEALLOC(ret, aParam))
38:                     case F-T-T ∨ T-T-T
39:                         macros.append(CALLER_DEALLOC(ret, aParam))
40:                     case T-F-T
41:                         macros.append(CALLEE_DEALLOC(ret, aParam))
42:             end if
43:         end for
44:     else// No needs to use post-deallocation macro
45:     end if
46: end procedure
```

Procedure 7.5 Generate Optimised C Code for Function *func*

Input: Function *func* at WyIL level
Output: The *list* of optimised C code using copy and deallocation analysers

```
1: Variables
2:     CopyAnalyser: Copy elimination analyser
3:     DeallocAnalyser: De-allocation analyser
4:     list: a list of optimised C code
5: end Variables
6: procedure CODE_OPTIMISE(func, list)
7:     list := []
8:     // Beginning function signature
9:     list.append("return_type ")// Function return type
10:    list.append("function_name(")// Function name
11:    for each param in func do// Function parameters
12:        type ← type of param from func function declaration
13:        list.append("type param,")// Append param
14:        if param is Array then// Add extra passed parameter
15:            list.append("size_t param_size,")
16:            list.append("bool param_dealloc,")
17:        end if
18:    end for
19:    list.append("){")// Ending function signature
20:    // Beginning variable declaration
21:    vars = variable tables from func variable declaration
22:    for each var in vars do
23:        type ← type of var
24:        list.append("type var;")
25:        if var is Array then
26:            list.append("size_t var_size = 0;")
27:            list.append("bool var_dealloc = false;")// Add deallocation flag
28:        end if
29:    end for// Ending variable declaration
30:    // Beginning function body
31:    for each code in function body do// Generate Optimised Code
32:        switch code do
33:            case Array Generator
34:                list.append(ARRAYGENERATOROPT(code, func))
35:            case Array Assignment
36:                list.append(ASSIGNMENTOPT(code, func))
37:            case Function Call
38:                list.append(FUNCTIONCALLOPT(code, func))
39:            case Return
40:                list.append(RETURNOPT(code, func))
41:            case Default// Generate Naive Code
42:                list.append(GENERATECODE(code, func))
43:    end for
44:    list.append("}")// Ending function body
45:    return list
46: end procedure
```

Array Generator Optimisation A new array generator creates an array *var* of given size with an initial value. We first use pre-deallocation macro to free array variable and assigns true flag to the de-allocation flag of array variable *var* using the following pre- and post macros

```
#define PRE_DEALLOC(var) // Pre−deallocation macro
({
  if(var_dealloc){free(var); var=NULL; var_dealloc=false;}
})
```

```
#define NEW_ARRAY_POST(var)
({
  var_dealloc=true;
})
```

Example 7.2 *Consider an assignment a = [2;3] where variable a is an array of size 3, and the value of each array item is 2. The code optimiser generates the below code*

```
PRE_DEALLOC(a); // Empty 'a' if 'a_dealloc' is true
NEW_1DARRAY(a, 2, 3, int64_t);// a = [2;3]
NEW_ARRAY_POST(a); // a_dealloc = true
```

Assignment Optimisation The optimisations of an assignment consist pre-deallocation macro, the copy of right variable and post-deallocation macro. Our optimiser does not deal with primitive typed assignment (integer/boolean) because it is made by call-by-value and optimised automatically by stack memory management.

```
#define PRE_DEALLOC(lhs)
({
  if(lhs_dealloc){free(lhs); lhs=NULL; lhs_dealloc=false;}
})
```

```
#define ADD_DEALLOC_POST(lhs, rhs)
({
  lhs_dealloc = true;
})
```

```
#define TRANSFER_DEALLOC_POST(lhs, rhs)
({
  lhs_dealloc = rhs_dealloc;
  rhs_dealloc = false;
})
```

Procedure 7.6 Generate Optimised C Code for Assignment *code* in *func*

Input: Assignment *code* in function *func* at WyIL level
Output: A *list* of optimised assignment code

```
1: Variables
2:     CopyAnalyser: Copy elimination analyser
3:     DeallocAnalyser: De-allocation analyser
4: end Variables
   // Produce optimised assignment code
5: procedure OptimiseAssignment(code, func)
6:     list = []
7:     lhs = left variable of code
8:     rhs = right variable of code
9:     list.append("PRE_DEALLOC(lhs)")// Pre-Deallocation Macro on lhs
10:    if CopyAnalyser.isCopyRemoved(rhs, code, func) then
11:        // Assignment without copy
12:        list.append(" lhs = rhs; lhs_size = rhs_size; ")
13:    else// Assignment with copy
14:        list.append(" lhs = COPY(rhs); lhs_size = rhs_size; ")
15:    end if
16:    macro ←DeallocAnalyser.choosePostDealloc(code, func)
17:    if macro == ADD_DEALLOC then
18:        list.append(" ADD_DEALLOC_POST(lhs, rhs) ")
19:    else// TRANSFER_DEALLOC
20:        list.append(" TRANSFER_DEALLOC_POST(lhs, rhs) ")
21:    end if
22:    return list
23: end procedure
```

Before an array assignment, deallocation analyser applies `PRE_DEALLOC` macro on left variable to empty its value, as above. Then the assignment statement itself can be optimised by copy analyser to remove the copy of right variable, and also apply `ADD_DEALLOC_POST` or `TRANSFER_DEALLOC_POST` post code to make changes of left and right variables' flag values after the assignment. Algorithm 7.6 shows the code optimisation on an assignment.

Example 7.3 *Consider an assignment* `a = b` *where variable* a *and* b *are arrays. The code optimiser generates an assignment with copy*

```
PRE_DEALLOC(a);
a = COPY(b);
ADD_DEALLOC_POST(a, b);
```

or an assignment without copy

```
PRE_DEALLOC(a);
a = b;
TRANSFER_DEALLOC_POST(a, b);
```

Function Call Optimisation The optimisation of a function call includes

- The code before a function call, including pre-deallocation of return variable and copying parameters
- Actual function call, including actual and copied parameters and deallocation flag value,
- Post-deallocation of parameters and return variable

Procedure 7.7 Variable name for array parameter at *index* of *code* in *func*

Input: Parameter at *index* of function call *code* in function *func* at WyIL level
Output: Temporary variable *name*

```
1: Variables
   map: Global Hash map stores the name of a temporary variable at index of
   code in func
2: end Variables
3: // Get the name of temporary variables used in code and func
4: procedure VarStore(index, code, func)
5:     name ← map.lookup(index, code, func)
6:     if name == NULL then// Make a new variable name
7:         fparam = name of formal parameter at position index in called
                    function of code
8:         name = "tmp_" + fparam
9:         suffix = 0
10:        while name is used in func do
11:            name = "tmp_" + fparam + "_" + suffix// Append suffix
12:            suffix++
13:        end while
14:        // Include name to map
15:        map.add((index, code, func) ↦ name)
16:    end if
17:    return name
18: end procedure
```

Our code optimisation focuses on array types, and does not need to have extra work on primitive typed parameters because C language takes call-by-value as default action to pass the values of those built-in types to the called function.

In addition, our code optimisation requires temporary variables to store the copied parameters at a function call. As the names of temporary variables should be different from any existing variable, we thus introduce *VarStore* to

keep track of all temporary variable names and avoid naming conflicts (see Algorithm 7.7).

Consider the called function *func*(*a*, *b*) as an example. The copy of array parameter at index of 1 would be *tmp_b*. If such a name is used in function *func*, then we use *tmp_b_0*, *tmp_b_1*, etc

Procedure 7.8 Generate code before function call *code* in *func*

Input: Function call *code* in function *func* at WyIL level
Output: A *list* of code before a function call

```
1:  Variables
    CopyAnalyser: Copy elimination analyser
    VarStore: A variable set stores the names of temporary variables
2:  end Variables
3:  // Generate pre-deallocation code of a function call
4:  procedure OPTIMISE_PREFUNCTIONCALL(code, func)
5:      list = []
6:      ret = function return variable at code
7:      if ret != NULL AND ret is an Array then
8:          list.append("PRE_DEALLOC(ret)")// Pre-Deallocation Macro on ret
9:      end if
10:     params = parameter list of code
11:     // Check each actual parameter param at code
12:     for index ← 0; index < |params|; index++ do
13:         param = params[index]
14:         if (param is an Array AND
15:           ¬CopyAnalyser.ISCOPYREMOVED(param, code, func)) then
16:             // Copying of param is needed
17:             tmp ←VarStore(index, code, func)// Temporary variable tmp
18:             // Store the copy of param with temporary variable tmp
19:             list.append("void* tmp = COPY(param);" )
20:         end if
21:     end for// Ending declaration
22:     return list
23: end procedure
```

Algorithm 7.8 shows the procedure of generating the code before an optimised function call. Firstly, the code optimiser gets the function return *ret* if provided, and then applies pre-deallocation macro to safely empty its value. Secondly, it goes through each actual parameter *param*, and if the parameter is an array and the copy is needed from copy analysis, declares an additional block-scope temporary variable *tmp*, which is obtained from *VarStore*, to store the copied function parameter.

Procedure 7.9 Generate actual function call *code* in *func*

Input: Function call *code* in function *func* at WyIL level
Output: A *list* of actual function call code

```
1: Variables
   CopyAnalyser: Copy elimination analyser
   DeallocAnalyser: De-allocation analyser
   VarStore: A variable set stores the names of temporary variables
2: end Variables
3: // Optimise actual function call
4: procedure OPTIMISE_ACTUALFUNCTIONCALL(code, func)
5:     list = []
6:     callee_name = get the name of called function at code
7:     ret = function return variable at code
8:     // Beginning of a function call
9:     if ret is NOT NULL then
10:        list.append(" ret = callee_name( ")
11:    else
12:        list.append(" callee_name( ")
13:    end if
14:    for index ← 0; index < |params|; index++ do
15:        param = params[index]
16:        if param is an Array then
17:            param_size = get size of array param
18:            if CopyAnalyser.isCopyRemoved(param, code, func) then
19:                // Pass param with false flag
20:                list.append(" param, param_size, false ")
21:            else
22:                macro ←DeallocAnalyser.choosePostMacro(param,
                       code, func)
23:                tmp ←VarStore(index, code, func)// Temporary variable
24:                // Pass copied parameter tmp
25:                if macro == CALLEE then
26:                    list.append(" tmp, param_size, true ")
27:                else// CALLER
28:                    list.append(" tmp, param_size, false ")
29:                end if
30:            end if
31:        else
32:            list.append(" param ")// Pass param without optimisation
33:        end if
34:    end for
35:    list.append(" ); ")// Ending of a function call
36:    return list
37: end procedure
```

Algorithm 7.9 shows the procedure of generating optimised actual function call. The code optimiser uses copy analysis results and post-deallocation macros, to produce the optimised function call, including the name of called function *callee_name*, parameter list and return variable if provided.

The parameter list includes all the actual parameters. We pass primitive typed parameters to called function without any extra optimisation, because they are copied and managed automatically by C language run-time. But for array-typed parameters, we need to have three arguments: array variable, size variable and the value of deallocation flag. The array variable can be actual parameter *param* or temporary variable *tmp*, depending on the needs of copying. Size variable is the size of actual parameter. And the deallocation flag is false by default, but if `CALLEE` macro is used, the flag is true.

Procedure 7.10 Generate optimised code after function call *code* in *func*

Input: Function call *code* in function *func* at WyIL level
Output: A *list* of code after a function call

```
1: Variables
   DeallocAnalyser: De-allocation analyser
   VarStore:A variable set stores the names of temporary variables
2: end Variables
3: procedure OPTIMISE_POSTFUNCTIONCALL(code, func)
4:     list = []
5:     ret = function return variable at code
6:     for index ← 0; index < |params|; index++ do
7:         if param is An Array then
8:             macro ←DeallocAnalyser.choosePostMacro(param, code, func)
9:             if ret is an Array then
10:                switch macro do
11:                    case RETAIN
12:                        list.append(" RETAIN_DEALLOC_POST(ret, param) ")
13:                    case RESET
14:                        list.append(" RESET_DEALLOC_POST(ret, param) ")
15:                    case CALLER
16:                        tmp ← VarStore(index, code, func)
17:                        list.append(" CALLER_DEALLOC_POST(ret, tmp) ")
18:                    case CALLEE
19:                        list.append(" CALLEE_DEALLOC_POST(ret, tmp) ")
20:            else if macro == CALLER then
21:                list.append(" free(tmp); ")// Free extra copy
22:            end if
23:        end if
24:    end for// Ending post macro
25:    return list
26: end procedure
```

Algorithm 7.10 shows the procedure of generating optimised code after a function call. The code optimiser goes through each array parameter, picks up its post-deallocation macro type from deallocation analyser, and inserts

the corresponding code (see the below macros) to make changes of run-time de-allocation flags between function return and parameters in the post state of a procedure call.

```
#define RETAIN_DEALLOC_POST(ret, param)
({
  ret_dealloc = true;
})
```

```
#define RESET_DEALLOC_POST(ret, param)
({
  if(ret != param){
    ret_dealloc = true;
  }else{
    ret_dealloc = param_dealloc;
    param_dealloc = false;
  }
})
```

```
#define CALLER_DEALLOC_POST(ret, tmp)
({
  if (ret != tmp) {free(tmp);}
  ret_dealloc = true;
})
```

```
#define CALLEE_DEALLOC_POST(ret, tmp)
({
  ret_dealloc = true;
})
```

These post code extracts from our de-allocation macros to set the deallocation flag of array typed function return and parameters after the call. In case of primitive typed returns, we do not apply our post code because those variables do not have flags. But since *CALLER* macro makes an extra copy of parameter and the called function does not return it, we therefore include a free statement to release temporary copy and avoid the memory leaks.

Example 7.4 *Consider a function call $d = func(a, b, c)$. Called function func returns an array variable d, and array variables a, b and c are passed parameters of function func. And then we use our analysis to determine the deallocation macros for parameters a, b and c and use* `CALLEE_DEALLOC`, `CALLER_DEALLOC` *and* `RETAIN_DEALLOC` *macros respectively. Then our code generator produces the below code:*

```
{
  PRE_DEALLOC(d);// Empty array 'd'
  // Copied parameters
  void* tmp_a_0 = COPY(a);
  void* tmp_b_0 = COPY(b);
  // Do not need to copy c;
  // Actual function call code
  d = func(
           tmp_a_0, a_size, true, // Pass copied 'a'
           tmp_b_0, b_size, false, // Pass copied 'b'
           c, c_size, false // Pass 'c' wihtout copy
           );
  // Post code
  CALLEE_DEALLOC_POST(d, tmp_a_0);
  CALLER_DEALLOC_POST(d, tmp_b_0);
  RETAIN_DEALLOC_POST(d, c);
}
```

Return Optimisation Apart from return variable, the code optimiser produces a list of PRE_DEALLOC macros to free the allocated memory space for all local array-typed variables and function parameters. We do not free return variable because it will be returned to caller site.

```
// Function 'func' may change 'b' array and may return 'b' array
// If not, return new array 'c'
function func(int[] b, int num) -> int[]:
    int[] c = [0;3] // c[0] = 0
    if num > 10:
        b[0] = num
        return b
    else:
        return c
// Method 'main'
method main(System.Console sys):
    int[] b = [2;3] // b[0] = 2
    int[] tmp = func(b, 11) // function call
    b = tmp // b[0] = tmp[0] = 11
    assert b[0] == 11
    sys.out.println(b[0])
    b = func(b, 65536)  // function call
    sys.out.println(b[0])
    assert b[0] == 65536
```

Listing 7.4: Example Whiley program

Example 7.5 *Consider the example 7.1 again. The Whiley source code is shown in Listing7.4. We use code optimisation to produce the code of function func and method main by:*

- *Eliminating unnecessary copies with copy analysis,*

- *Inserting pre and post-deallocation macros into the generated code by using de-allocation analysis*

The code generator works with copy and deallocation analysers, and the procedure of code optimisation starts with function *func* and then moves on to method *main*, and performs on each line of code in each function. The code generator checks copy analysis to delete or keep the copying, and deallocation analyser to choose the macros for copy optimised C code.

```
1  // function func(int[] b, int num) -> int[]:
2  int64_t* func(int64_t* b, size_t b_size, bool b_dealloc,
3                int64_t num, size_t* _size){
4    int64_t* _6 = NULL;size_t _6_size = 0;bool _6_dealloc = false;
5    int64_t*  c = NULL;size_t  c_size = 0;bool  c_dealloc = false;
6    NEW_1DARRAY(_6, 0, 3, int64_t);//arraygen %6 = [0; 3] : int[]
7    NEW_ARRAY_POST(_6); // _6_dealloc=true;
8    PRE_DEALLOC(c);
9    c = _6; c_size = _6_size;//assign c = %6  : int[]
10   TRANSFER_DEALLOC_POST(c, _6); // c_dealloc=true, _6_dealloc=false
11   if(num<=10){goto blklab0;}
12   b[0] = num; //update b[0] = num
13   PRE_DEALLOC(c);// c_dealloc = true
14   PRE_DEALLOC(_6);// _6_dealloc = false
15   *_size = b_size; // Update return array size to call-by-reference '_size'
16   return b;//return b
17 blklab0:;
18   PRE_DEALLOC(b);// b_dealloc = false
19   PRE_DEALLOC(_6);// _6_dealloc = false
20   *_size = c_size;
21   return c;//return c
22 }
```

Listing 7.5: Code snippet of copy optimised function *func* (comments: deallocation flag or WyIL code)

Function *func* Includes an extra *b_dealloc* to indicate if parameter *b* can be freed by function *func* or not. In the first statement, we create a new array variable *_6*, and assign true *_6_dealloc* flag using `NEW_ARRAY_POST` macro.

The next assignment in line 8 writes array *_6* to variable *c* without copies because *_6* has no uses and becomes dead after this program point.

In line 9, we place a branch depending on *num* value.

- *num* > 10: we update and return parameter *b* and thus need to free all the other local variables using `PRE_DEALLOC` macro. In this case, we will only free variable *c* because *_6* and *c* are aliased to the same array and only variable *c* has true flag.

- $num \leq 10$: we return a new array c and thus need to use PRE_DEALLOC macro parameter b and temporary variable *_6*. And we will not free *_6* because *_6* has a false flag. The de-allocation of variable b will depend on the value of passed *b_dealloc* parameter.

Before each return, we update the size of output array to call-by-reference parameter *_size* so that the array size can be passed back to caller site.

```
1  int main(int argc, char** args){
2    int64_t*  _5=NULL; size_t  _5_size=0; bool  _5_dealloc=false;
3    int64_t*   b=NULL; size_t   b_size=0; bool   b_dealloc=false;
4    int64_t*  _8=NULL; size_t  _8_size=0; bool  _8_dealloc=false;
5    int64_t* tmp=NULL; size_t tmp_size=0; bool tmp_dealloc=false;
6    int64_t* _18=NULL; size_t _18_size=0; bool _18_dealloc=false;
7    NEW_1DARRAY(_5, 2, 3);// arraygen %5 = [2; 3] : int[]
8    NEW_ARRAY_POST(_5); // _5_dealloc = true;
9    PRE_DEALLOC(b);
10   b = _5; b_size = _5_size;//assign b = %5  : int[]
11   TRANSFER_DEALLOC_POST(b, _5);
12   //b_dealloc = true, _5_dealloc = false
13   {//invoke ()%8) = func(b, 11)
14     PRE_DEALLOC(_8);
15     _8 = func(b, b_size, false, 11, &_8_size)// Pass 'b' without copy
16     RESET_DEALLOC_POST(_8, b);
17   }// _8_dealloc = true, b_dealloc = false
18   PRE_DEALLOC(tmp);
19   tmp = _8; tmp_size = _8_size;//assign tmp = _8  : int[]
20   TRANSFER_DEALLOC_POST(tmp, _8);
21   // tmp_dealloc = true, _8_dealloc = false
22   PRE_DEALLOC(b);
23   b = tmp; b_size = tmp_size;//assign b = tmp  : int[]
24   TRANSFER_DEALLOC_POST(b, tmp);// b_dealloc = true, tmp_dealloc =
       false
25   ASSERT(b[0] == 11);
26   printf("%"PRId64"\n", b[0]);
27   {//invoke (%18) = func(b, 65536)
28     PRE_DEALLOC(_18);
29     _18 = func(b, b_size, false, 65536, &_18_size);
30     RESET_DEALLOC_POST(_18, b);
31   }//_18_dealloc = true, b_dealloc = false
32   PRE_DEALLOC(b);
33   b = _18; b_size = _18_size;//assign b = _18  : int[]
34   TRANSFER_DEALLOC_POST(b, _18);//b_dealloc = true, 18_dealloc =
       false
35   ASSERT(b[0] == 65536);
36   printf("%"PRId64"\n", b[0]);
37   PRE_DEALLOC(b);// b_dealloc = true
38   PRE_DEALLOC(tmp);// tmp_dealloc = false
39   PRE_DEALLOC(_5);// _5_dealloc = false
40   PRE_DEALLOC(_8);// _8_dealloc = false
41   PRE_DEALLOC(_18);// _18_dealloc = false
42   exit(0);//return
43 }
```

Listing 7.6: Code snippet of copy cptimised method *main* (comments: deallocation flag)

Method main Firstly creates and assigns a new array to variable *b* without copies, so *b_dealloc* is true. In the next, we make a function call to *func* so apply PRE_DEALLOC macro to empty *_8*, which is not executed because of false *_8_dealloc* value. We use RESET_DEALLOC post-deallocation macro on variable *b* because:

- Passed parameter *b* may be updated and returned by called function *func*;
- Variable *b* becomes dead after the call as the assignment in line 20 over-writes variable *b* at method *main*

So the function call with expanded macro shows as follows:

```
1  { //_8_dealloc=false
2    if(_8_dealloc){free(_8); _8=NULL; _8_dealloc=false;}
3    // b_dealloc = true
4    _8 = func(b, b_size, false, 11, &_8_size);
5    // 'b' will not be freed by 'func'
6    if(_8 != b){ // _8 points to a new array
7      _8_dealloc = true;
8    }else{ // _8 and b point to the same array
9      _8_dealloc = b_dealloc;// _8_dealloc = _b_dealloc = true
10     b_dealloc = false; // _b_dealloc = false
11   }
12 }
```

Since the deallocation flag *b_dealloc* is passed as false value to called function *func*, *b* will not be free by function *func*. Because *b* and *_8* are the same array, reset macro will transfer the flag from *b* to *_8*.

In the next two assignments (line 19 to 26), we move array *_8* from *tmp* to *b* by using transfer macro, so does the deallocation flag value.

In the second call (line 24 to 28) we also use reset macro similarly to assign the flag from parameter *x* to target variable *_18*, and then over-writes array *b* with variable *_18*.

Throughout the entire main method, we have only one copy of array *b*, made by NEW_1DARRAY macro. Although 6 different variables alias to this array, only variable *b* has true flag. Therefore, by using PRE_DEALLOC macro we can restrict the memory de-allocation on variable *b* only, and free the shared array without double free errors.

7.2.4 Choosing Fixed-Size Integers

Whiley programming language provides two types of integers: `int` and `byte`. By default, we uses signed 64-bit integers (`int64_t`) to store the value of each integer/array, regardless of its domain in the program. For **byte** typed integers/arrays, because its value range always falls within 0 and 255, we use unsigned 8 bit integers (`uint8_t`) to store its value.

Procedure 7.11 Choosing Suitable Integer type

Input: Integer Variable *var* of function *func*
Output: Fixed-size integer *type* for *var* suggested by our bound analyser

```
1: Variables
2:     type: Fixed-sized Integer types (int16_t, int32_t, int64_t, uint16_t,
   uint32_t, uint64_t)
3:     MAX(type): Maximal value of type
4:     MIN(type): Minimal value of type
5: end Variables
   // Use bound result to choose fixed-width integer type
6: procedure ChooseIntegerType(var, func)
7:     d = domain(var)
8:     lower = d.getLower()// Get lower bound
9:     upper = d.getUppser()// Get upper bound
10:    if lower ≥ 0 then// Unsigned integer
11:        if upper ≤ MAX(uint16_t) then
12:            return uint16_t
13:        else if upper ≤ MAX(uint32_t) then
14:            return uint32_t
15:        else
16:            return uint64_t
17:        end if
18:    else// Signed integer
19:        if MIN(int16_t) ≤ lower AND upper ≤ MAX(int16_t) then
20:            return int16_t
21:        else if MIN(int32_t) ≤ lower AND upper ≤ MAX(int32_t) then
22:            return int32_t
23:        else
24:            return int64_t
25:        end if
26:    end if
27: end procedure
```

Once those domains can be statically estimated by our bound analysis, we can use inferred lower and upper bound to choose suitable integer type (see Algorithm 7.11). For example, the lower bound of an integer variable has only positive value (no negative value), and then we can use unsigned integer types.

Then by checking the upper bound with maximal value of each type, we can determine integer size, e.g. unsigned 16 bits (`uint16_t`) or unsigned 32 bits (`uint32_t`) to hold its value. Currently our bound analysis supports below integer types and its ranges:

Table 7.1: Supported fixed-width integer type and value range

Integer type	Description	$[Min \dots Max]$
`int16_t`	Signed integer with exactly 16 bits	$[-(2^{15}-1)\dots 2^{15}-1]$
`int32_t`	Signed integer with exactly 32 bits	$[-(2^{31}-1)\dots 2^{31}-1]$
`int64_t`	Signed integer with exactly 64 bits	$[-(2^{63}-1)\dots 2^{63}-1]$
`uint16_t`	Unsigned integer with exactly 16 bits	$[0\dots 2^{16}-1]$
`uint32_t`	Unsigned integer with exactly 32 bits	$[0\dots 2^{32}-1]$
`uint64_t`	Unsigned integer with exactly 64 bits	$[0\dots 2^{64}-1]$

Some compiler can generate the most efficient implementation for a basic typed integer but its size varies depending on platform and program request. For example, `int` integers has a range of sizes varying from 16 to 64 bits as long as it holds the requested value. Thus, we may have 32-bit on one compiler and 64-bit on another even if the same processor is used.

Using fixed-size integers in our generated code results in consistent and portable memory usage across different platforms because fixed-sized integers always use the exact width of memory space as indicated on the name (`uint16_t` integer takes only 16 bits wide of memory). We therefore can estimate the required memory space and ensure the program is able to execute in a limited memory embedded system.

Our analyser performs bound analysis on each function call: propagating input bounds from caller site to called procedure, extracting range constraints and then inferring the bounds using fixed-point iteration along with widening operator. Lastly, our analyser stores the bound results of each call separately and takes union of all function calls to produce aggregated final bounds for all

integer variables, including input parameters and return value.

Our code optimisation may alias some variables due to copy elimination, and those aliasing also changes the variable bounds. Our bound analyser goes through all aliasing variables at final stage, makes the union of bounds and updates the bounds of all aliasing variables. With this information, our code generator can select a fixed-size integer for each variable within its range.

```
1  function func(int[] b, int num) -> int[]:
2      int[] c = [0;3] // c[0] = 0
3      if num > 10:
4          b[0] = num
5          return b// Function 'func' may change and return 'b' array
6      else:
7          return c// If not, return new array 'c'
8
9  method main(System.Console sys):
10     int[] b = [2;3] // b[0] = 2
11     int[] tmp = func(b, 11) // function call
12     b = tmp // b[0] = tmp[0] = 11
13     assert b[0] == 11
14     sys.out.println(b[0])
15     b = func(b, 65536)  // function call
16     sys.out.println(b[0])
17     assert b[0] == 65536
```

Listing 7.7: Example Whiley program

Example 7.6 *Consider the example 7.7 again. We enable bound analysis to find the matching integer types for each target variable. Method main makes two function calls at line 12 and 15 and over-writes variable b twice. Each call passes different value of parameter num and results in different bounds of function func. Our bound analyser examines all calls and takes union of bounds to produce final results for function func, and apply the resulting bounds to use integer types in the code.*

```
1  // d(b) = [2..2] d(num) = [11..11] d(return) = [0..11]
2  function func(int[] b, int num) -> int[]:
3    int[] c = [0;3] // d(c) = [0..0]
4    if num > 10:
5      // num > 10 => d(num)=[11..11]
6      b[0] = num // d(b) = [2..11]
7      return b // d(b) = [2..11]
8    else:
9      // Unreachable block
10     // num <= 10 => d(num)=[empty..empty]
11     return c // d(c) = [0..0]
```

Listing 7.8: Bound inference on 1st function call $func(b,\ 11)$ (comments: inferred bounds)

1st Function Call *func*(b, *11*) (see Listing 7.8) take array b and integer *num* as inputs, and extracts the constraints from condition in line 4 for IF and ELSE blocks, and starts fixed-point iteration to infer the bounds in each block. Given input bounds $d(b) = [2 \dots 2]$ and $d(num) = [11 \dots 11]$, we have:

- IF block ($num > 10$):

$$d(num) = d(num) \cap [11 \dots \infty] = [11 \dots 11]$$

  The update statement in line 6 changes the domain of variable b

$$d(b) = d(b) \cup [11 \dots 11] = [2 \dots 2] \cup [11 \dots 11] = [2 \dots 11]$$

  The above domains are feasible so make IF block reachable.

- ELSE block ($num \leq 10$):

$$d(num) = d(num) \cap [-\infty \dots 10] = [11 \dots 10] = \emptyset$$

$$d(c) = [0 \dots 0]$$

  Domain $d(num)$ is not feasible so makes ELSE block unreachable.

The return variables are aliased to b and c (see return statements in line 7 and 11) even although ELSE block is unreachable, variable c is aliased with function return. So we can obtain the domain of return variable as the union bounds of all aliasing variables, and then update the resulting domain to all aliasing variables. The output domain of 1st function call *func*(b, *11*) is

$$d(return) = d(b) \cup d(c) = [2 \dots 11] \cup [0 \dots 0] = [0 \dots 11] = d(b) = d(c)$$

```
1  method main(System.Console sys):
2    int[] b = [2;3]
3    int[] tmp = func(b, 11) // d(tmp) = d(return) = [0..11]
4    b = tmp // d(b) = d(tmp) = [0..11]
5    assert b[0] == 11
6    sys.out.print_s("b[0] = ")
7    sys.out.println(b[0]) // d(b) = [0..11]
8    b = func(b, 65536)
9    assert b[0] == 65536
10   sys.out.print_s("b[0] = ")
11   sys.out.println(b[0])
```

Listing 7.9: Bound propagation on 1st function call *func*(b, *11*) (comments: inferred bounds)

Method main *Main* (see Listing 7.9) propagates domain $d(return)$ back to variable *tmp* at caller , and then assignment in line 4 passes the domain from variable *tmp* to b. So we have below domains

$$d(b) = d(tmp) = d(return) = [0 \dots 11]$$

Before the 2nd function call, $d(b)$ is updated to $d(b) = [0 \dots 11]$

```
1 // d(b) = [0 .. 11] d(num) = [65,536 .. 65,536] d(return) = [0 .. 65,536]
2 function func(int[] b, int num) -> int[]:
3   int[] c = [0;3] // d(c) = [0 .. 0]
4   if num > 10:
5     b[0] = num // d(b) = [0 .. 65,536]
6     return b // d(b) = [0 .. 65,536]
7   else:
8     return c // d(c) = [0 .. 0]
```

Listing 7.10: Code snippet of bound inference on 2nd function call *func* (comments: inferred domain)

2nd Function Call *func*(b, *65536*) (see Listing 7.10) takes $d(b) = [0 \dots 11]$ and $d(num) = [65,536 \dots 65,536]$ as input bounds , and produces output the following bounds:

- IF block ($num > 10$):

  $$d(num) = d(num) \cap [11 \dots \infty] = [65,536 \dots 65,536]$$

  The updated domain of b is

  $$d(b) = d(b) \cup [65,536 \dots 65,536] = [0 \dots 11] \cup [65,536 \dots 65,536]$$
  $$= [0 \dots 65,536]$$

  The above domains are feasible so make IF block reachable.

- ELSE block ($num \leq 10$):

  $$d(num) = d(num) \cap [-\infty \dots 10] = [65,536 \dots 10] = \emptyset$$
  $$d(c) = [0 \dots 0]$$

  Domain $d(num)$ is not feasible so make ELSE block unreachable.

Therefore, we combine the bounds of variable b and c to produce the domain of return value, and then update resulting domains to variable b and c.

$d(return) = d(b) \cup d(c) = [0 \dots 0] \cup [0 \dots 65,536] = [0 \dots 65,536] = d(b) = d(c)$

Final Bounds Function *func* combines the results of 1st and 2nd function calls to produce the bounds for function *func*. The domains are summarised as follows.

Table 7.2: Final Domains of Function *func*

Domain(var)	1st Call	2nd Call	Final bounds	Integer Type
d(b)	$[0 \dots 11]$	$[0 \dots 65,536]$	$[0 \dots 65,536]$	`uint32_t`
d(num)	$[11 \dots 11]$	$[65,536 \dots 65,536]$	$[11 \dots 65,536]$	`uint32_t`
d(c)	$[0 \dots 11]$	$[0 \dots 65,536]$	$[0 \dots 65,536]$	`uint32_t`
d(return)	$[0 \dots 11]$	$[0 \dots 65,536]$	$[0 \dots 65,536]$	`uint32_t`

Our code generator bases on the final inferred bound results (see Table 7.2) to choose a specific fixed-size type for each variable, and generates C code of function *func*. For example, we use unsigned 32-bit integers to store array c because its domain falls within $[0 \dots 2^{32} - 1]$. And we also can use `uint32_t` types for input parameters b and *num* since their ranges are within unsigned 32-bit integers.

```
1  // d(b) = [2..65,536] d(num) = [11..65,536]
2  uint32_t* func(uint32_t* b, size_t b_size, uint32_t num,
3    size_t* _size){
4    uint32_t* _6=NULL;
5    size_t    _6_size=0; bool _6_dealloc = false;
6    uint32_t*  c=NULL;
7    size_t     c_size=0; bool  c_dealloc = false;
8    NEW_1DARRAY(_6, 0, 3, uint32_t);
9    PRE_DEALLOC(c);
10   c = _6; c_size = _6_size;
11   TRANSFER_DEALLOC_POST(c, _6);
12   if(num<=10){goto blklab0;}
13   b[0] = num;
14   PRE_DEALLOC(c);
15   PRE_DEALLOC(_6);
16   *_size = b_size;
17   return b;
18 blklab0:;
19   PRE_DEALLOC(b);
20   PRE_DEALLOC(_6);
21   *_size = c_size;
22   return c;
23 }
```

Listing 7.11: Code snippet of function *func* using fixed-sized integers (comments: inferred bounds)

Method main We will illustrate the bounds of a variable may be changed due to aliasing effects caused by the copy optimisation.

```
1  // Main method in our example
2  method main(System.Console sys):
3    int[] b = [2;3] // d(b) = [2..2]
4    int[] tmp = func(b, 11) // d(tmp) = [0..11]
5    b = tmp // d(b) = d(tmp) = [0..11]
6    assert b[0] == 11
7    sys.out.print_s("b[0] = ")
8    sys.out.println(b[0])
9    b = func(b, 65536) // d(b) = [0..65536] = d(tmp)
10   assert b[0] == 65536
11   sys.out.print_s("b[0] = ")
12   sys.out.println(b[0])
```

Listing 7.12: Code snippet of bound inference on method *main* (comments: inferred domain at each program point)

Example 7.7 *Assignment in line 4 at Method main assigns array tmp to b. In copy removed code, because the copy is taken out at the assignment, array b is aliased to tmp. Because of variable aliasing, we need to use an extra step to produce final bounds.*

Table 7.3: Final bounds of copy eliminated method *main*

Variable	Domain	Integer Type
d(_5)	$[0 \dots 65,536]$	`uint32_t`
d(b)	$[0 \dots 65,536]$	`uint32_t`
d(_8)	$[0 \dots 65,536]$	`uint32_t`
d(tmp)	$[0 \dots 65,536]$	`uint32_t`
d(_18)	$[0 \dots 65,536]$	`uint32_t`

Once copy analyser optimises and removes the unused copies at method *main*, our bound analysis starts the bound inference procedure and meanwhile, keeps track of variable aliasing sets, where all aliased variables are store in the same set.

At the final phase of bound inference, our analyser goes through each variable in the same aliased set to take the union of all aliasing variable domains, and then use the union domain to update all relevant variables' bounds. Therefore, we have consistent variable domain to fit into maximal and minimal values of all aliased variables.

Consider our example again. The domain results of copy eliminated code are listed as follows.

Table 7.3 shows the bound analyser produces consistent bound results for copy optimised code. After the copies are removed, variable b is aliased to four variables: *_5*, *_8*, *tmp* and *_18*.

Variable *_5* is the target of array generator code at line 2 in example 7.12. The array generator code creates an array of size 3, and initialises all array elements with 2, and then assigns to variable *_5*. So domain $d(\_5)$ is $[2 \ldots 2]$.

Variable *_8* and *_18* are the return at 1st and 2nd function calls respectively. From previous section, we know the domain of 1st function return $d(\_8)$ is $[0 \ldots 11]$ and is updated to variable *tmp* and b.

$$d(tmp) = d(\_8) = [0 \ldots 11] = d(b)$$

The domain of 2nd return *_18* is $[0 \ldots 65,536]$ and overwrites variable b, so the final domains are

$$d(b) = d(\_18) = [0 \ldots 11] \cup [0 \ldots 65,536] = [0 \ldots 65,536]$$

Since the copy eliminated code removes copies at all assignments and aliases all variables, so the analyser takes union of bounds and update the domains of all aliasing variables: b, *_5*, *tmp* and *_18*.

The final bounds are updated to the below domain

$$d(b) = d(tmp) = d(\_5) = d(\_18) = [0 \ldots 65,536]$$

```
1  int main(int argc, char** args){
2    uint32_t*  _5=NULL; size_t  _5_size=0; bool  _5_dealloc=false;
3    uint32_t*   b=NULL; size_t   b_size=0; bool   b_dealloc=false;
4    uint32_t*  _8=NULL; size_t  _8_size=0; bool  _8_dealloc=false;
5    uint32_t* tmp=NULL; size_t tmp_size=0; bool tmp_dealloc=false;
6    uint32_t* _18=NULL; size_t _18_size=0; bool _18_dealloc=false;
7    // arraygen %5 = [2; 3]
8    NEW_1DARRAY(_5, 2, 3); _5_dealloc = true;
9    //assign b = %5  : int[]
10   PRE_DEALLOC(b);
11   b = _5; b_size = _5_size;
12   TRANSFER_DEALLOC_POST(b, _5); // b_dealloc = true, _5_dealloc =
       false
13   //invoke ()%8) = func(b, 11)
14   {
15     PRE_DEALLOC(_8);
16     _8 = func(b, b_size, false, 11, &_8_size)// Pass 'b' without copy
17     RESET_DEALLOC_POST(_8, b); // _8_dealloc = true, b_dealloc =
       false
18   }
19   //assign tmp = _8  : int[]
20   PRE_DEALLOC(tmp);
21   tmp = _8; tmp_size = _8_size;
22   TRANSFER_DEALLOC_POST(tmp, _8);// tmp_dealloc = true, _8_dealloc
       = false
23   //assign b = tmp  : int[]
24   PRE_DEALLOC(b);
25   b = tmp; b_size = tmp_size;
26   TRANSFER_DEALLOC_POST(b, tmp);// b_dealloc = true, tmp_dealloc =
       false
27   ASSERT(b[0] == 11);
28   printf("%"PRId64"\n", b[0]);
29   //invoke (%18) = func(b, 65536)
30   {
31     PRE_DEALLOC(_18);
32     _18 = func(b, b_size, false, 65536, &_18_size);
33     RESET_DEALLOC_POST(_18, b);// _18_dealloc = true, b_dealloc =
       false
34   }
35   //assign b = _18  : int[]
36   PRE_DEALLOC(b);
37   b = _18; b_size = _18_size;
38   TRANSFER_DEALLOC_POST(b, _18); // b_dealloc = true, 18_dealloc =
       false
39   ASSERT(b[0] == 65536);
40   printf("%"PRId64"\n", b[0]);
41   PRE_DEALLOC(b);// b_dealloc = true
42   PRE_DEALLOC(tmp);// tmp_dealloc = false
43   PRE_DEALLOC(_5);// _5_dealloc = false
44   PRE_DEALLOC(_8);// _8_dealloc = false
45   PRE_DEALLOC(_18);// _18_dealloc = false
46   //return
47   exit(0);
48 }
```

Listing 7.13: Copy eliminated code with integer bound inference results on method *main*

Chapter 8

Benchmarks for Sequential Programs

The use of value semantics in Whiley functional programming language introduces expensive overheads of array copying, when array is large. Also, the generated code, if we naively translate WyIL code into sequential C code, has memory leaking issues and thus can not scale to larger problem sizes.

Our code optimiser analyses a Whiley program at WyIL level and offsets above inefficiency at code generation phase to produce efficient C code that can run fast and for long. Our copy analyser eliminates unused copies to reduce copying overheads and our deallocation analyser chooses and inserts macros at appropriate program points to avoid memory leaks and errors, so the resulting code has fewer overheads and leaks than naive one and thus speed up the execution.

Our static bound analysis is *disabled* for all benchmarks, because our benchmark program varies the problem sizes at runtime by taking command line arguments or a text file, whose value can not be statically estimated by our analysis to give out precise integer ranges and types.

This chapter goes through a series of benchmark programs to illustrate effectiveness of our code optimisation. Each benchmark program is firstly compiled into WyIL code. Then our code generator takes WyIL code as input,

translates into C code with/without copy and de-allocation analysers, and give four kinds of C11-compatible implementations:

- Naive code (`N`) is translated from WyIL code with no optimisation.
- Naive and de-allocated code (`N+D`) is translated from WyIL code and optimised with de-allocation analyser only.
- Copy-eliminated code (`C`) is translated from WyIL code and optimised with just copy elimination analyser.
- Copy-eliminated and de-allocated code (`C+D`) is translated from WyIL code and optimised with both copy elimination and de-allocation analysers.

Each implementation for 10 times on one problem size, and average the execution time of 10 runs. The performance metric includes

- Memory leaks of each implementation are detected by `Valgrind` (Nethercote and Seward, 2007) and summed up 4 kinds of memory leaks (definitely, indirectly, possibly and still reachable losses).
- Speedup of copy elimination is the execution rate of naive code over copy eliminated code $\frac{N}{C}$
- Speedup of combined optimisation is the execution rate of naive + de-allocated code over copy eliminated + de-allocated code $\frac{N+D}{C+D}$

All benchmarks are conducted on Ubuntu machine (i7-4770 CPU @ 3.40GHz and 16 GB memory), and compiled into executable by GCC compiler (version 5.4.0) with `O3` optimisation flag.

8.1 Micro-Benchmarks

The micro-benchmark consists of 5 Whiley programs to test code optimisations and measure the performance of generated C code. The benchmark suite includes `Reverse` (see Appendix B.1) *TicTacToe* (see Appendix B.2) *MergeSort*

(see Appendix B.4), *BubbleSort* (see Appendix B.3) and *MatrixMulti* (see Appendix B.5) programs.

Each test case takes command line arguments as input to vary the array size of benchmark programs. In each case, we choose three sizes to measure the memory leaks and execution time. Note *TicTacToe* program varies the number of repeats, rather than the size of game board, for bench-marking.

Table 8.1: Memory leaks (bytes) of micro-benchmarks

		Memory Leaks (bytes)			
Test Case	Problem Size	**N**	**N + D**	**C**	**C + D**
Reverse	100,000	4,800,416	0	1,600,408	0
	1,000,000	48,000,424	0	16,000,416	0
	10,000,000	480,000,432	0	160,000,424	0
TicTacToe	100,000	276,000,296	0	204,000,288	0
	200,000	552,000,296	0	408,000,288	0
	300,000	828,000,296	0	612,000,288	0
BubbleSort	1,000	32,408	0	8,400	0
	10,000	320,416	0	80,408	0
	100,000	3,200,424	0	800,416	0
MergeSort	1,000	320,376	0	80,368	0
	10,000	640,648	0	160,544	0
	100,000	961,144	0	240,776	0
MatrixMult	$1,000 \times 1,000$	112,000,464	0	24,000,456	0
	$2,000 \times 2,000$	448,000,464	0	96,000,456	0
	$3,000 \times 3,000$	1,008,000,464	0	216,000,456	0

Memory Leaks Table 8.1 shows that, on our benchmark suite, our deallocation analysis effectively avoids memory leaks on both naive and copy eliminated code for all test cases. Also, the copy elimination alone can effectively remove copies in all test cases, and avoid all unnecessary copies in four cases (at least):

Reverse, *BubbleSort*, *MergeSort* and *MatrixMult*. Note in each case, there are minor and constant amounts of memory leaks, e.g. 424 bytes in *Reverse* case, which do not grow with problem sizes, because our program needs to allocate some extra memory space to store the values of command line arguments.

```
function reverse(int[] arr) -> int[]:
  int i = |arr|
  int[] r = [0; |arr|]
  while i > 0 where i <= |arr| && |r| == |arr|:
    int item = arr[|arr|-i]
    i = i - 1
    r[i] = item
  return r
```

Listing 8.1: Reverse program

Reverse program uses two arrays (*arr* and *r*) to run function *reverse* (see Listing 8.1). Because each array is declared as signed 64-bit integers (`int64_t`), we can get the number of arrays used in the program as estimates of memory leaks.

Consider the array size of 1×10^7 as an example. Each array takes up 80 MB, and the memory leaks in Table 8.1 show our copy elimination analysis reduces *six* arrays down to only *two*, and thus removes all redundant array copies. Leaks in *Reverse* program also have a linear relation with array sizes, and then we can get $3.3 \times 10^8 = (16GB/48bytes)$ as the estimated maximal size of naive *Reverse* code. We can choose 1×10^8, 2×10^8 and 3×10^8 as array sizes to benchmark speed-ups.

```
function bubbleSort(int[] items) -> int[]:
  int length = |items|
  int last_swapped = 0 // Until no items is swapped
  while length > 0:
    last_swapped = 0
    int index = 1
    while index < length:
      if items[index-1] > items[index]:
        int tmp = items[index-1]
        items[index-1] = items[index]
        items[index] = tmp
        last_swapped = index
      index = index + 1
    length = last_swapped// Skip the remaing items as they are ordered.
  return items
```

Listing 8.2: Bubble sort program

BubbleSort program creates and sorts one array of `int64_t` type. Consider the array size of 1×10^5, or 0.8 MB in memory. The leaking results show our

copy elimination analysis removes all copies and keeps only one array to do bubble sorting. We choose 1×10^5, 2×10^5 and 3×10^5 as benchmark levels to measure the speed-ups of code optimisation.

```
function sortV1(int[] items, int start, int end)->int[]:
  if (start+1) < end:
    int pivot = (start+end) / 2
    int[] lhs = Array.slice(items,start,pivot)
    lhs = sortV1(lhs, 0, pivot)
    int[] rhs = Array.slice(items,pivot,end)
    rhs = sortV1(rhs, 0, (end-pivot))
    ...
    // Merge 'lhs' and 'rhs' arrays
    while i < (end-start) && l < (pivot-start)
              && r < (end-pivot):
     ...
  return items
```

Listing 8.3: Merge sort program

Similarly, in *MergeSort* program our copy elimination can also remove all unnecessary copies and reduce four arrays down to one.

Table 8.1 show the memory leaks are not severe in *MergeSort* and *BubbleSort* programs, so we can benchmark speed-up on larger array sizes. Since the memory leaks in both cases increase linearly with array size, we can predict that naive *MergeSort* code runs out of memory at array size of $5.0 \times 10^7 = 16(GB)/320(bytes)$ as an estimate of memory leaks. Therefore, we can set benchmark levels to 1.0×10^7, 2.0×10^7 and 3.0×10^7 for both *MergeSort* and *BubbleSort* cases.

```
function mat_mult(int[] a, int[] b, int[] data, int width, int height)
    -> (int[] c):
  int i = 0
  while i < height:
    int j = 0
    while j < width:
      int k = 0
      int sub_total = 0
      while k < width:
        sub_total=sub_total+a[i*width+k]*b[k*width+j]
        k = k + 1
      data[i*width+j] = sub_total
      j = j + 1
    i = i + 1
  return data
```

Listing 8.4: Matrix multiplication program

MatrixMult program creates three matrices of `int64_t` type and represents each matrix with a single dimensional array. So in the case of $1,000 \times 1,000$,

each matrix amounts to 8 MB. The results show our copy elimination removes all redundant copies but keeps only three necessary matrices to compute matrix multiplication. Without memory deallocation the naive C code has server leaks. For example, when matrix size is increased up-to $4,000 \times 4,000$, the naive *MatrixMult* code amounts to 17.92 GB and exceeds the memory limits and causes system breakdown.

Table 8.2: Average execution time (seconds) of micro-benchmarks

		Implementation				Speed-up	
Test Case	Problem Size	**N**	**N + D**	**C**	**C + D**	$\frac{N}{C}$	$\frac{N+D}{C+D}$
Reverse	1×10^8	0.903	1.195	0.351	0.371	2.58	3.22
	2×10^8	1.744	1.735	0.694	0.694	2.51	2.50
	3×10^8	2.609	2.608	1.015	1.027	2.57	2.54
TicTacToe	100,000	0.241	0.193	0.156	0.118	1.54	1.64
	200,000	0.412	0.353	0.277	0.225	1.49	1.57
	300,000	0.615	0.517	0.405	0.342	1.52	1.51
BubbleSort	100,000	6.659	6.627	6.634	6.616	1.00	1.00
	200,000	26.399	26.396	26.418	26.398	1.00	1.00
	300,000	59.358	59.372	59.377	59.364	1.00	1.00
MergeSort	1×10^7	0.078	0.077	0.040	0.035	1.95	2.19
	2×10^7	0.148	0.149	0.046	0.067	3.21	2.21
	3×10^7	0.196	0.191	0.063	0.073	3.13	2.62
MatrixMult	$1,000 \times 1,000$	1.28	1.27	1.29	1.39	1.00	0.92
	$2,000 \times 2,000$	19.3	19.2	19.1	19.1	1.01	1.01
	$3,000 \times 3,000$	47.9	47.7	47.9	48.0	1.00	0.99

Execution Time and Speed-up Table 8.2 shows that our de-allocation macro (N+D) does not slow down the execution of naive code in all cases. Copy elimination (C) and the combined optimised (C+D) code both increases speed-ups with array sizes in *Reverse*, *TicTacToe* and *MergeSort*.

In conclusion, our combined optimised (C+D) code runs as fast as copy eliminated code in *Reverse* and *TicTacToe*, but runs slower in *MergeSort* case. Our de-allocation macro takes up time to free allocated memory and thus introduces delays in execution. Since the time in merge sort case is comparatively small, the delays become more significant than other two cases.

The flat speed-ups in *BubbleSort* and *MatrixMult* cases require further profiling to find out performance bottlenecks. By using `gprof` tool, we can know naive *BubbleSort* code spends almost 100% time on sorting and swapping array items. Likewise, naive *MatrixMult* code takes 99% time to calculate the products of rows and columns, and spends only 0.1% on array copying. Since their computation dominates the overheads of array copies and memory deallocation, our code optimisation has little effects on speed-ups.

8.2 Case Study: Cash Till

The cash till test case simulates a series of transactions in a cash register. Typical transaction is: a customer buys one product and gives out certain amounts of money, and then the cash till calculates the correct amount of change and returns to the customer.

```
1  function buy(Cash till, Cash given, int cost) -> Cash:// Compute
       changes
2    if total(given) >= cost:
3      Cash|null change = calculateChange(till,total(given) - cost)
4      if change != null:
5        till = add(till,given)// Receive customer's payment
6        till = subtract(till,change)// Return changes to customer
7    return till
8  public method main(System.Console console):// Main entry point
9    int repeat = 0
10   while repeat < max:
11     Cash till = Cash() // Start with empty cash till
12     if repeat%2==1:// Change every 2 iterations to avoid the same results
13       till = [5,3,3,1,1,3,0,0]// Start with none−empty cash till
14     // now, run through some sequences...
15     till = buy(till,Cash([ONE_DOLLAR]),85)//Cash: $1, Cost: $0.85
16     till = buy(till,Cash([ONE_DOLLAR]),105)//Cash: $1, Cost: $1.05
17     till = buy(till,Cash([TEN_DOLLARS]),5)//Cash: $10, Cost: $5
18     till = buy(till,Cash([FIVE_DOLLARS]),305)//Cash: $5, Cost: $3.05
19     console.out.println_s(toString(till))// Result cash in till
20     repeat = repeat + 1
```

Listing 8.5: Code Snippets of Cash Till Whiley Program

Listing 8.5 shows *CashTill* benchmark program (full version sees Appendix B.6). The cash till calculates the amount of change to be returned to customer with customer's payment and current cash in the till, and produces the output of each transaction, e.g. the till may be short of cash change, or customer's payment is insufficient for the cost.

The benchmark program registers one cash till and initialises its change in the till, and then runs through 4 kinds of transactions and prints out final cash in the till. Each benchmark repeats for a number of times, which is passed from command line argument, and switches initial change of cash till every iteration.

Table 8.3: Memory leaks (bytes) of cash till

	Implementation			
Repeats	N	N + D	C	C + D
100	983,902,456	0	737,819,248	0
200	1,967,804,856	0	1,475,638,448	0
300	2,951,707,256	0	2,213,457,648	0

Memory Leaks Table 8.3 show our de-allocation analysis avoids all leaks and, without our deallocation macros, naive or copy-eliminated C code has severe memory leaks and fails to run large-scaled problems. Also, the memory leaked in *Cashtill* case grow linearly with problem sizes, so we can roughly estimate the amount of leaks and the maximal problem size for our benchmark machine. For example, running naive C code at 1,600 repeats would accumulate up to 15.74 GB leaks ($16 \times 0.984 = 15.74$) and uses up all the system memory of 16 GB. Note that 512 MB is reserved for Ubuntu OS.

We choose 1,000 to 2,000 as benchmark sizes to increase the execution time and measure the speed-ups.

Table 8.4: Average execution time (seconds) of cash till (OOM: out-of-memory)

	Implementation				Speed-up	
Repeats	**N**	**N + D**	**C**	**C + D**	$\frac{N}{C}$	$\frac{N+D}{C+D}$
1,000	9.43	7.99	6.27	5.28	1.50	1.51
1,200	11.51	9.59	7.49	6.33	1.54	1.51
1,400	48.49	11.21	8.72	7.42	5.56	1.51
1,600	OOM	12.80	9.99	8.42		1.52
1,800	OOM	14.35	28.83	9.48		1.51
2,000	OOM	15.96	OOM	10.54		1.51

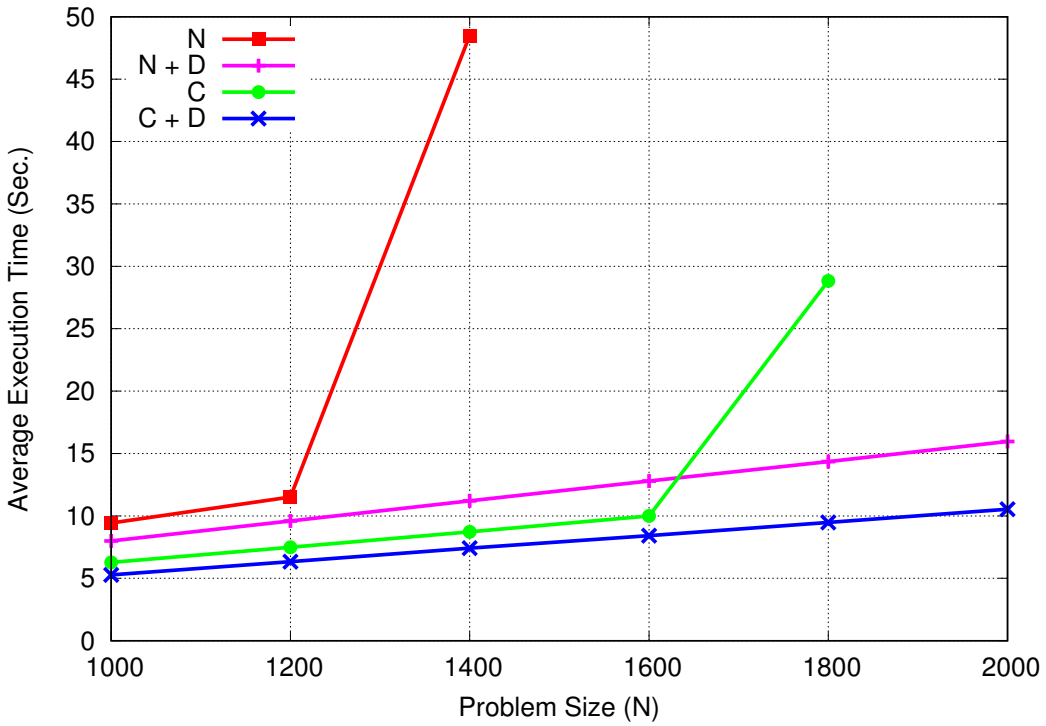


Figure 8.1: Execution time graph of cash till test case

Execution Time and Speedup Table 8.4 also show that, naive or copy-eliminated C code has out of memory conditions in large-scaled problem sizes and fails the execution. As the size reaches to maximal repeat number $(1,400)$,

the increased memory leaks cause naive code to greatly slows down the execution. The slow-down may be caused by page thrashing (Denning, 1968). When naive or copy eliminated code runs out of physical memory, it requests the access to store data on disk. But it takes time to swap data from memory to disk and the computation time also suffers by slow disk access. Therefore, due to slow paging, naive or copy eliminated Cash Till code runs several orders of magnitude slower than deallocated code on 1,400 and 1,800 problem sizes respectively.

Figure 8.1 shows both de-allocated only (N+D) and combined optimised (C+D) code increase the execution time linearly with problem size. That means, the cash till program with de-allocation optimisation has linear time complexity $O(n)$ where n is the number of coins in the till, so the time needed for processing transactions in a cash till depends on the number of coins the till has.

In conclusion, our de-allocation analysis not only stops memory leaks effectively but also make the code run fast and for long. In particular, the combined optimisation (C+D) can produce the fastest execution and steadily gain $1.51x$ speed-up over de-allocated code (N+D).

8.3 Case Study: Coin Game

Dynamic programming (Cormen, 2009) is a typical divide and conquer technique to optimise the program. First, it breaks down a problem into smaller problems and, solves each of sub-problems and then store or "memorise" the solutions for later use. When the same sub-problem occurs, the program can look up the previous solution without computation and speed up the execution.

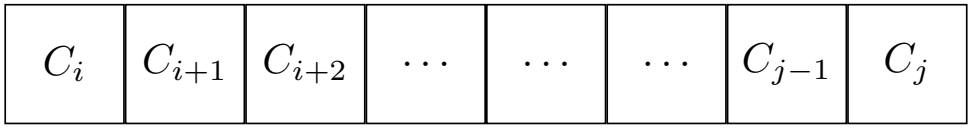


Figure 8.2: A line of coin array C_n

Coin-In-A-Line Game is an example of dynamic programming. Suppose N coins are placed in a line from left to right, and each coin is worth $C_i = i\%5$ and its value ranges from 0 up-to 4, as shown in Figure 8.2

Assume we have two players: Alice and Bob, and Alice plays the game first. Alice and Bob take turns to pick one coin up either from start or end of the line. Winner of the game is to collect the most golds. Our goal is to develop a game strategy to help Alice win the game using dynamic programming.

The dynamic programming strategy uses $MOVES[i][j]$, a two-dimensional array, to store the maximal coin values that Alice can collect from coin C_i and C_j. Because both Alice and Bob are keen to win, Alice or Bob will choose her/his best pick, make the move and leave the minimal value coins for the opponent.

We can split the move $MOVES[i][j]$ in below cases and find the best one to maximise Alice's total gain:

- Assume Alice picks up C_i. Bob needs to choose C_{i+1} and C_j, and Alice's next move depends on Bob's decision.
  - If Bob chooses C_{i+1}, then Alice has to pick C_{i+2} or C_j.
  - If Bob chooses C_j, then Alice has to pick C_{i+1} or C_{j-1}.

  Bob also chooses the coin that will leave Alice to have fewer gains. So Alice can only collect the coins:

$$C_i + min(MOVES[i+2][j],\ MOVES[i+1][j-1]) \qquad (8.1)$$

- Assume Alice picks up C_j. Bob needs to choose between C_i and C_{j-1}.
  - If Bob picks C_i, then Alice has to pick C_{i+1} or C_{j-1}.
  - If Bob picks C_{j-1}, then Alice has to pick C_i or C_{j-2}.

  Bob also leaves Alice with minimal value coins. So Alice can collect the coins:

$$C_j + min(MOVES[i+1][j-1],\ MOVES[i][j-2]) \qquad (8.2)$$

From Equations 8.1 and 8.2, we can have the optimal move for Alice:

$$MOVES[i][j] = max(C_i + min(MOVES[i+2][j],\ MOVES[i+1][j-1]),$$
$$C_j + min(MOVES[i+1][j-1],\ MOVES[i][j-2]))$$

We also can divide the coin game into N steps, and then solve each step sequentially and keep track of all moves. By doing so, we can re-use the results from the previous step and reduce expensive re-computation overheads.

```
1  // Use dynamic programming to find all moves for Alice
2  function findMoves(int[] moves, int n, int[] coins) -> int[]:
3    int s = 0// s: step
4    while s < n: // Find the optimal 'move[i][j]' in 's' step
5      int i = 0 // coin[i]
6      while i < n -s: //
7        int j = i + s // coin[j] (remaing coin from 'i+s' upto 'n')
8        int y = moves[(i + 1)*n + (j - 1)] // moves[i+1][j-1]
9        int x = moves[(i + 2)*n + j]// moves[i+2][j]
10       int z = moves[i*n + (j - 2)]// moves[i][j-2]
11       moves[i*n+j] = Math.max(coins[i] + Math.min(x, y),
                                 coins[j] + Math.min(y, z))
12       i = i + 1// End of i,j loop
13     s = s + 1// End of s loop
14   return moves
15 method main(System.Console sys):
16   ....
17   int[] coins = [0;n]
18   int i = 0
19   while i < n:
20     coins[i] = i % 5 // Coin array is [0,1,2,3,4,0,1,2,3,4...]
21     i = i + 1
22   // Increase 'moves' array to (n+2)*(n+2)
23   // so that if/else branches at 'findMoves' function can be avoided
24   int[] moves = [0;(n+2)*(n+2)]
25   moves = findMoves(moves, n, coins)
26   int sum_alice = moves[n-1]
```

Listing 8.6: Coin game Whiley program

Listing 9.4 shows coin game Whiley program (full version sees Appendix B.7). We optimise *findMoves* function to avoid any branch using below steps:

- We extend *MOVES* array size to $(N + 2) \times (N + 2)$ to accommodate all array access, e.g. $i + 2$ or $j - 2$, without needing of bound checks.
- We use below macros(Anderson, 2005) to find the maximum and minimum without branching
  - `max(a, b) = a ^ ((a ^ b) & - (a < b))`
  - `min(a, b) = b ^ ((a ^ b) & -(a < b))`

Table 8.5: Memory leaks (bytes) of coin game

	Implementation			
Problem Size	**N**	**N + D**	**C**	**C + D**
100	335,968	0	84,664	0
1,000	32,152,776	0	8,040,672	0
10,000	3,201,520,784	0	800,400,680	0

Memory Leaks Table 8.5 shows enabling de-allocation analysis can effectively avoid memory leaks and make the program run on larger scaled problem. Also, we find out that naive or copy eliminated code increases memory leaks linearly with problem size, and we can check if any memory space is wasted in the implementation. For example, if the problem size is $10,000$, then the program at least uses (0.08 + 800) MB as we declare *coins* as 1D array of `int64_t` type, and *moves* as 2D array of `int64_t` type. Results show copy eliminated code does not have any extra copy whereas naive code makes three times of unnecessary copying.

Table 8.6: Average execution time (seconds) of coin game test case

	Implementation				Speed-up	
Problem Size	**N**	**N + D**	**C**	**C + D**	$\frac{N}{C}$	$\frac{N+D}{C+D}$
10,000	0.739	0.736	0.360	0.333	2.06	2.21
20,000	2.995	2.977	1.400	1.400	2.14	2.13
25,000	OOM	4.749	2.253	2.222		2.14
30,000	OOM	OOM	3.13	3.17		
40,000	OOM	OOM	5.75	5.76		

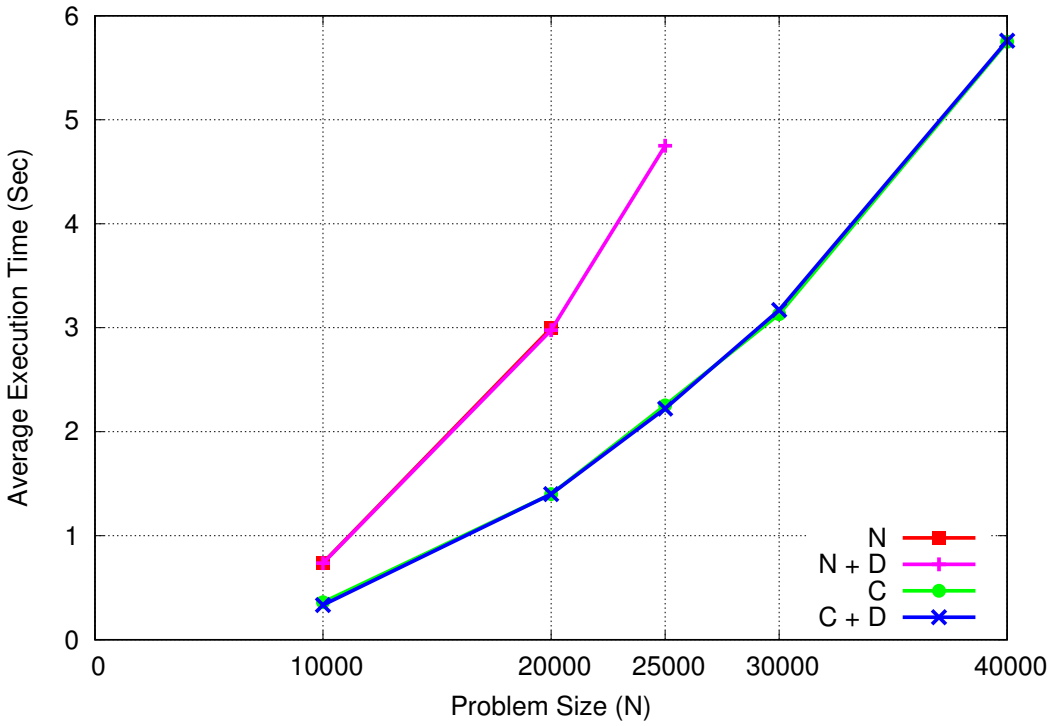


Figure 8.3: Execution time graph of coin game

Execution time and Speed-up Table 8.6 shows copy eliminated (C) and combined optimised (C+D) code both gain steady and scalable speed-ups with problem size. Speed-ups $\frac{N}{C}$ show that eliminating unnecessary copies increases the speed of program. Speed-ups $\frac{N+D}{C+D}$ show extra de-allocation code does not slow down but make the program runs faster. Figure 8.3 shows copy eliminated (C) and combined optimised (C+D) coin game code both have the fast and scalable execution whereas naive (N) or de-allocated only code (N+D) requires lots of memory space so fails to run on large problem sizes.

In conclusion, copy elimination improves the performance of coin game program and the combined optimised code (C+D) has the fastest execution and runs for long. The optimised code runs in quadratic time $O(n \times n)$, where n is the total number of coins. The quadratic time complexity is because the program iterates n steps and the first step processes at most n coins. Thus, the program takes $O(n \times n)$ space to enumerate all possible moves.

8.4 Case Study: LZ77 Algorithm

LZ77 algorithm (Ziv and Lempel, 1977) allows to reduce the redundancy of sequential data, and compress to a list of encoded matches for better storage. And to save compression time, the encoder maintains a fixed-sized sliding window to limit the maximal number of searched strings and time. Upon decoding, the decompression restores each encoded match into a corresponding string and appends it to output.

In this case, we will investigate the procedure and optimisation of LZ77 compression and de-compression separately. Each program is translated and optimised into different C code, and compiled by `GCC` and bench-marked on `Ubuntu` machine (`Intel` i7-4770 CPU @ 3.40GHz and 16 GB). We will show the memory leaks of each generated code and speed-ups from our code optimisation.

Table 8.7: Offset-length pairs encoded in LZ77 compression of sample string

Position	Input: 'AACAACABCABAAAC'		Output Pair
	Lookup Array	Encoded Array	(offset, length)
0	∅	A ACAACABCABAAAC	(0, 'A')
1	A	A CAACABCABAAAC	(1, 1)
2	AA	C AACABCABAAAC	(0, 'C')
3	AAC	AACA BCABAAAC	(3, 4)
7	AACAACA	B CABAAAC	(0, 'B')
8	AACAACAB	CAB AAAC	(3, 3)
11	AACAACABCAB	AA AC	(11, 2)
13	AACAACABCABAA	AC	(12, 2)

8.4.1 LZ77 Compression

LZ77 compression program implements the Lempel-Ziv 77 algorithm to compress an input string into a list of encoded numeric pairs. The LZ77 encoder splits the string from current position into *lookup array* that occurs earlier, and an *encoded array*. It matches the string of encoded array with lookup one, to find the best match which has the longest size or reaches the slide window of 256. The found match is then encoded to an `offset-length` pair, where *offset* is distance from current position to the match and *length* is match length. Once encoded, the match is moved to lookup array. In the case that no match could be found, the target character is encoded as a single match, e.g. (*0*, 'A').

Once the input string is encoded to above offset-length matches, as shown in Table 8.7, the encoder writes out each match to a byte array as output. By doing so, the decoder reads each item from compressed array, i.e. offset-length pair, decode it to a string. For example, the match (*12*, *2*) at position 13 shows that the longest match has the offset of 12 and length of 2. Given such a match information, the decoder starts from position 13 and goes back 12 characters to position 1, and then copies 2 characters AC from existing decompressed string and inserts to the end of output array.

In the case that the match has null offset value and no repetitive words is found for a specific word, e.g. (*0*, 'A') the decoder takes out the value of *length* item and appends to output array. The decoder repeatedly decompresses all matches and restores them to the original input string.

8.4.1.1 LZ77 Compression using Append Array

The LZ77 compressor takes an uncompressed data array as input, and produces as output a compressed byte array. The compressor continuously searches for repeated occurrence/match (see function *match*) until it finds the longest one (see function *findLongestMatch*), and then encodes as an offset-length match. If not found, then the compressor encodes as a special match. The matches

are appended to output array using function *append*.

```
// Match type stores 'offset−length'
type Match is ({nat offset, nat len} this)
// Find the length of a match from data array
function match(byte[] data, nat offset, nat end) -> (int length)
  ensures 0 <= length && length <= 255:
  nat pos = end
  nat len = 0
  while offset < pos && pos < |data|
    && data[offset] == data[pos] && len < 255:
    offset = offset + 1
    pos = pos + 1
    len = len + 1
  return len
// Find the longest match for 'pos' position
// data: uncompressed data array, pos : current position
function findLongestMatch(byte[] data, nat pos) -> (Match m):
  nat bestOffset = 0
  nat bestLen = 0
  int start = Math.max(pos - 255, 0)// Sliding window size of 255
  nat offset = start
  while offset < pos:
    int len = match(data, offset, pos)
    if len > bestLen: // Find the longest match
      bestOffset = pos - offset
      bestLen = len
    offset = offset + 1
  return {offset:bestOffset, len:bestLen} // Return a 'Match' object
// Append one byte to a byte array
function append(byte[] items, byte item) -> (byte[] nitems):
  nitems = [0b; |items| + 1]
  int i = 0// Make a copy of passed 'items' array
  while i < |items|:
    nitems[i] = items[i]
    i = i + 1
  nitems[i] = item
  return nitems
// Compress data array into output array
function compress(byte[] data) -> (byte[] output):
  nat pos = 0
  output = [0b;0]
  while pos < |data|:
    Match m = findLongestMatch(data, pos)
    // Encode the match to 'offset−length' pair
    // offset: distance to longest match, length: length of longest match
    byte offset = Int.toUnsignedByte(m.offset)
    byte length = Int.toUnsignedByte(m.len)
    if offset == 00000000b:// No match is found.
      length = data[pos] //Put the first byte of encoded array
      pos = pos + 1
    else:
      pos = pos + m.len// Skip the matched bytes
    // Write 'offset−length' pair to output array
    output = append(output, offset)
    output = append(output, length)
  return output
```

Listing 8.7: LZ77 compression Whiley program using append array

Listing 8.7 shows LZ77 compression Whiley program (full version sees Appendix B.8). Each function will be discussed as follows.

Function *findLongestMatch* Searches and returns the longest match. The search starts from current position backward to at most 255, so that the found match ($0 \sim 255$) can fit into a byte array without overflows. The function continuously increments and passes offset value to function *match* to find the match length for each offset and obtain the best match which has the longest length.

Function *match* Takes input string array *data* as input and returns the match length for a given position *pos* and offset value *offset*. Consider the match AC at position of 13 and offset value of 1. The match searching is:

$$\mathsf{pos} = 13 \quad \mathsf{offset} = 1 \quad \mathsf{data[offset]} = A \quad \mathsf{data[pos]} = A \quad \mathsf{len} = 1$$

$$\mathsf{pos} = 14 \quad \mathsf{offset} = 2 \quad \mathsf{data[offset]} = C \quad \mathsf{data[pos]} = C \quad \mathsf{len} = 2$$

The match does not continue because it reaches the limit of array size, and thus returns the length of 2.

Function *append* Makes a copy of input array and appends one item to the end of output array. Each call creates one array and copies each array items, and thus is slow and can be sped up by our pattern transform.

8.4.1.2 LZ77 Compression using Pre-allocate Array

Function *compress* starts with an empty array and then appends each offset-length pair to the output array. Because function *compress* can be matched with append array pattern, we can use the idea of our pattern transform (see in Definition 4.11) to replace slow array appending with efficient array update.

We can pre-allocate a larger array and resize the array to its actual size. Instead of initialising with an empty array, we create a larger array with over-estimated size using the number of loop iterations *loop_iters* and the number of append function calls n

$$arr\_size(output) = loop\_iters(pos) \times n = |data| \times 2$$

The number of loop iterations is bound to the length of input array and function *append* is invoked twice in each iteration. Therefore, we have the maximal size of output array $2 \times |data|$. Once all the compressed data are stored in the preallocated output array, and then we can shrink the array to actual size by using function *resize* as shown in the following program.

```
1  // Shrink the input array to the array of given array size
2  function resize(byte[] items, int size) -> (byte[] nitems)
3  requires |items| >= size
4  ensures |nitems| == size:
5    nitems = [0b; size]
6    int i = 0
7    while i < size:
8      nitems[i] = items[i]
9      i = i + 1
10   return nitems
11 // Compress in LZ77 algorithm using resize pattern
12 function compress(byte[] data) -> (byte[] output):
13   nat pos = 0
14   output = [0b;2*|data|]// Pre−allocates 2x input array size
15   int size = 0 // Actual array size
16   while pos < |data|:// Iterate each 'data' array item
17     Match m = findLongestMatch(data, pos)
18     byte offset = Int.toUnsignedByte(m.offset)
19     byte length = Int.toUnsignedByte(m.len)
20     if offset == 00000000b:
21       length = data[pos]
22       pos = pos + 1
23     else:
24       pos = pos + m.len
25     // Update output array with 'offset−length' pair
26     output[size] = offset
27     size = size + 1
28     output[size] = length
29     size = size + 1
30   // Reduce output array to actual size
31   output = resize(output, size)
32   return output
```

Listing 8.8: LZ77 compression Whily program using preallocated array

Listing 8.8 shows the transformed compression function and efficient preallocating and resizing array. Initially the output array is allocated with double the size of input array so that the array is big enough to place all pairs without needing to extend its capacity and check out-of-bound errors.

While encoding, we use lower-overhead and in-place array update, instead of slow array appending, to write out $offset - length$ pairs as output. Meanwhile, we use variable *size* to keep trace of actual array size so that we can shrink output array to final size and reduce the memory usage.

8.4.1.3 Benchmark Results

LZ77 compression program reads a text file as input, and produces a byte array of encoded matches. The benchmark Whiley program is translated into four kinds of code with append array and preallocate array.

Table 8.8: Memory leaks (bytes) of LZ77 compression

		Implementation			
	Problem Size	N	N + D	C	C + D
Append Array	M1x(1.58 kb)	278,322,116	0	1,217,518	0
	M2x(3.16 kb)	1,186,085,396	0	4,780,233	0
	M4x(6.32 kb)	4,889,942,900	0	18,946,735	0
Preallocate Array	M1x(1.58 kb)	273,529,702	0	20,205	0
	M2x(3.16 kb)	1,167,113,084	0	38,736	0
	M4x(6.32 kb)	4,814,446,504	0	75,798	0

Memory Leaks Table 8.8 shows that our de-allocation analysis can effectively avoid all the memory leaks on both transformed and untransformed LZ77 compression, and that copy elimination analysis can reduce 99% of leaks from naive code. That means, the naive code creates too many unneeded copies and does not delete memory when no longer used, and therefore accumulates a large sum of memory leaks.

However, pre-allocate array has better memory-saving effects on copy-reduced code as it avoids the memory leaks of the function call *append* which over-writes the output array without freeing the old array. Since function *append* no longer is used, the program using pre-allocate array has much lower memory leaks than the program using append array. All these leaks can be completely eliminated by using our de-allocation macros.

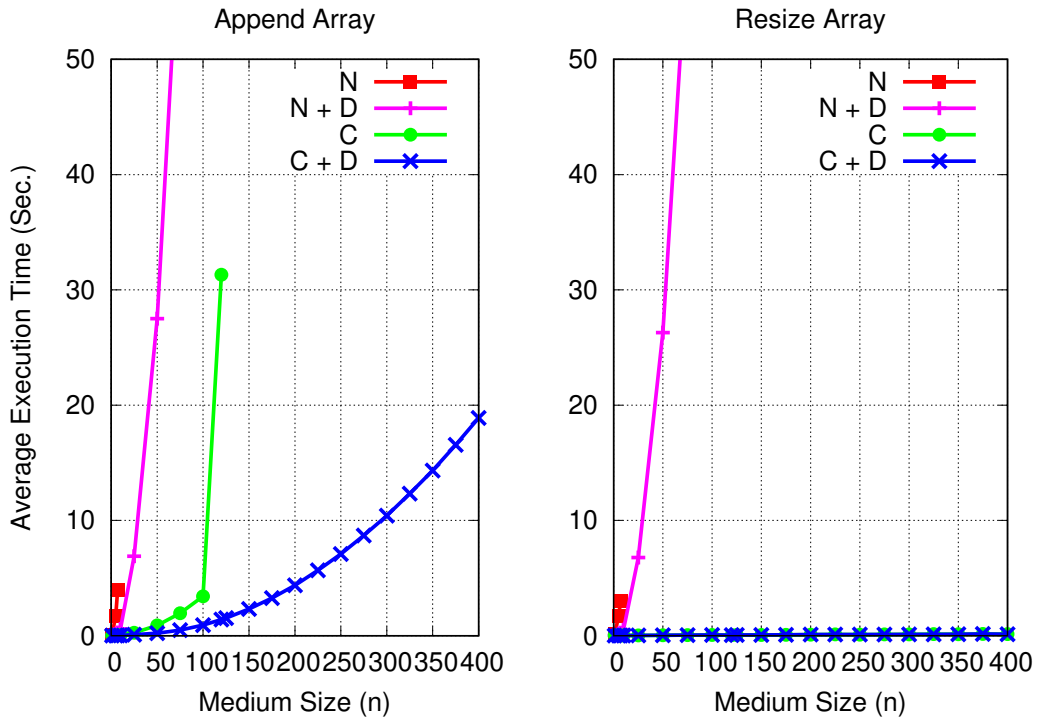


Figure 8.4: Execution time graph of LZ77 compression on medium sizes

Execution Time on Medium-Sized File We benchmark our code with a variety of medium-sized files, ranging from 1.5 KB to 404.7 KB. Appendix Table B.1 and Appendix Table B.2 show the results of LZ77 compression program using append array and pre-allocate array respectively.

Figure 8.4 shows both append and pre-allocate array programs can vary from minutes to few milliseconds and depend on whether the copies are eliminated or not. Due to severe memory leaks, the naive code stops execution at small $7x$ problem sizes.

The de-allocated-only (N+D) code runs faster than naive code, and can scale up to the largest problem size. However, it has the slowest execution. Copy-eliminated (C) code runs fast but encounters out-of-memory problems if we use append array function. And the combined optimised (C+D) code has the fastest execution and runs for long with both append and pre-allocate arrays. The de-allocated-only (N+D) code takes quadratic amount of time $O(n^2)$ where n is the array size of input data. Function *compression* goes

through n items of input data array and makes a function call to find the longest match. And each call requires the copying of input array, so the code has quadratic time complexity.

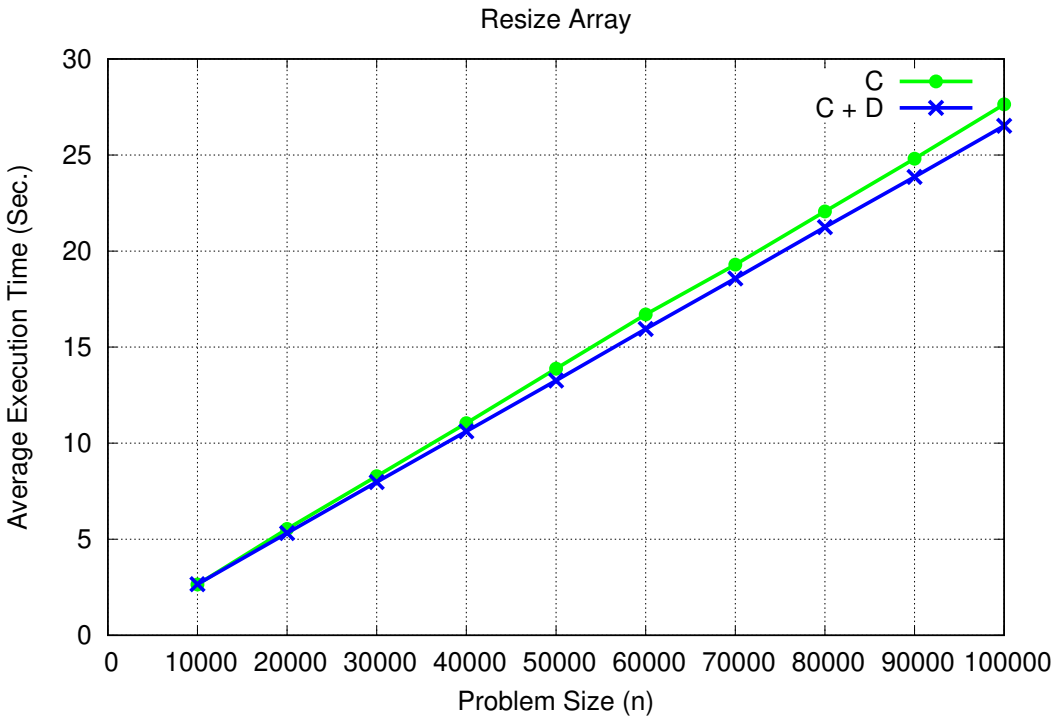


Figure 8.5: Execution time graph of LZ77 compression using pre-allocate array on large sizes

Execution Time on Large-Sized File The execution time of our benchmark results is too small to investigate the time complexity and may have measurement errors, e.g. copy eliminated + preallocate array. We will re-run benchmarks on large files from 10,000x magnitudes (15.8 MB) to 100,000x (158.1 MB) and investigate the efficiency. The detailed benchmark results are listed in Appendix Table B.3.

Figure 8.5 show, that combined optimised (C+D) and copy eliminated (C) code using pre-allocate array have similar speeds and run in linear time with file size. So we can conclude LZ77 compression using pre-allocate array reduces time complexity from quadratic $O(n^2)$ down to linear time $O(n)$, and thus gains large amounts of speed-ups and improves the program scalability.

8.4.2 LZ77 Decompression

LZ77 decompression program takes the compressed byte array as input, goes through each encoded pair and decodes the content to its original string and then appends it to output array. We have two kinds of implementations depending on the behaviour of array appending.

8.4.2.1 LZ77 Decompression using Append Array

Function *decompress* processes each pair of compressed array in order and checks if the pair is a match to restore the output string. For a no-match pair, we append the length, which contains only one item, to output array. For a match, we obtain the values of offset and length. Then the decoder goes back *offset* bytes from current position to read the specified number of bytes *len* and append to current end of output array.

```
1  // Append a byte to the byte array
2  function append(byte[] items, byte item) -> (byte[] nitems):
3    ensures |nitems| == |items| + 1:
4    nitems = [0b; |items| + 1]
5    int i = 0
6    while i < |items|:
7      nitems[i] = items[i]
8      i = i + 1
9    nitems[i] = item
10   return nitems
11 // Decompress input data array to original byte array
12 function decompress(byte[] data) -> (byte[] output):
13   output = [0b;0]
14   nat pos = 0
15   while (pos+1) < |data|:
16     // Get the pair
17     byte header = data[pos]
18     byte item = data[pos+1]
19     pos = pos + 2
20     if header == 00000000b:// For none−match pair
21       output = append(output, item)
22     else:// For match pair
23       int offset = Byte.toUnsignedInt(header) // Get offset
24       int len = Byte.toUnsignedInt(item) // Get length
25       // Go beack to 'offset' from current position
26       int start = |output| - offset
27       int i = start
28       // Read 'length' bytes and append to output array
29       while i < (start+len):
30         item = output[i] // Get one byte
31         output = append(output, item) // Append to output
32         i = i + 1
33   return output// Return the decompressed array
```

Listing 8.9: LZ77 decompression using one-by-one array appending

Listing 8.9 shows code snippet of LZ77 decompression using one-by-one array appending (full version sees Appendix B.9). The decoder still uses slow array appending (function *append*) to construct output array. Similarly, we can statically estimate the size of decompressed array as $128x$ magnitudes of the length of compressed array:

$$< \text{Length of decompressed array} >= 128\times < \text{Length of compressed array} >$$

because each match size may vary from 1 to 256 randomly. However, allocating such a big memory space is hard to implement. Instead, we choose Java-like *array list* implementation over array to optimise LZ77 decompression.

```
1  // If full, then double array size and store the data
2  function opt_append(byte[] items, nat item_len, byte new_item) ->
       byte[]:
3    if item_len < |items|: // 'items' array is large enough
4      items[item_len] = new_item// Have in-place array update
5    else:
6      // Copy 'items' array and append new item to the end of 'items' array.
7      byte[] nitems = [0b; |items|*2+1]
8      int i = 0
9      while i < |items|:
10       nitems[i] = items[i]
11       i = i + 1
12     nitems[i] = new_item
13     items = nitems
14   return items
15 // Decompress 'data' to byte array using array list
16 function decompress(byte[] data) -> (byte[] output):
17   byte[] items = [0b;0]
18   nat item_len = 0// Current item number in array list
19   nat pos = 0
20   while (pos+1) < |data|:
21     byte header = data[pos]
22     byte item = data[pos+1]
23     pos = pos + 2
24     if header == 00000000b:
25       // Append 'item' using array list
26       items = opt_append(items, item_len, item)
27       item_len = item_len + 1 //Increment 'item_len'
28     else:
29       int offset = Byte.toUnsignedInt(header)
30       int len = Byte.toUnsignedInt(item)
31       int start = item_len - offset
32       int i = start
33       while i < (start+len):
34         item = items[i]
35         // Append 'item' using array list
36         items = opt_append(items, item_len, item)
37         item_len = item_len + 1 //Increment 'item_len'
38         i = i + 1
39   // all done!
40   output = resize(items, item_len)// Shrink array into accurate length
41   return output
```

Listing 8.10: LZ77 Decompression using Array List

8.4.2.2 LZ77 Decompression using Array List

Array list dynamically grows the array to its double size when the array is full, and then manipulates the array using fast in-place update, and therefore it runs in constant time $O(1)$. We use array list to generate the output array and speed up LZ77 decompression.

As shown in Listing 8.10, function *decompress* includes variable *item_len* to keep track of current number of items stored in the array, and use it to check whether the array reaches its capacity and to decide re-allocating the array. So when item length is small than array size, the array is large enough for a new item, so we can use in-place array update to add this item and run in constant time $O(1)$.

In the case that array list is full, we create a new array with doubled its size, copy all items from old array and append the new item to the end of new array. This procedure is similar to array append and takes quadratic time complexity $O(n^2)$. By doing so, we reduce the occurrences of expensive array copies and make use of fast array update when populating output array.

The decoder iterates through each match in compressed array, and converts the match to a string and append to the output using array list. Once de-compression finishes, we can resize and shrink the output array to actual length and decrease memory usage, as shown in Listing 8.10 (full version sees Appendix B.10).

8.4.2.3 Benchmark Results

LZ77 decompress benchmark reads a compressed file as input, and decodes the content and produce a string (byte array) as output. We experiment the decompression with static or dynamic array appending separately to find out which one is much efficient.

Table 8.9: Memory leaks (bytes) of LZ77 decompression

		Implementation			
Append	Problem Size	N	N + D	C	C + D
Array	M1x(1.58 kb)	5,007,726	0	1,252,913	0
	M2x(3.16 kb)	20,012,850	0	5,004,577	0
	M4x(6.32 kb)	80,017,830	0	20,006,588	0
Array List	M1x(1.58 kb)	3,704,521	0	8,006	0
	M2x(3.16 kb)	14,767,380	0	15,214	0
	M4x(6.32 kb)	58,970,541	0	29,631	0

Memory Leaks Table 8.9 shows that our de-allocation analysis effectively avoids all the memory leaks in both naive and copy eliminated code. The leaks show naive code takes $O(n^2)$ space in both array and array list, and copy eliminated code grows $O(n)$ space in array and less $O(n)$ space in array list. Furthermore, in copy eliminated code array list can reduce the memory usage by an order of closely linear magnitude, which is difficult quantifying space complexity reduced by array list as the leaking data are proportional to problem sizes in logarithmic space $O(log_2 n)$.

In LZ77 decompression case, our copy elimination decreases quadratic amount of memory space down to linear space, and using array list over one-by-one array appending can further reduce the large amount of memory usages.

Execution Time and Speedup on Medium Sizes We benchmark our code and vary the sizes of compressed files, which are the outputs from LZ77 compression program. The detailed benchmark results are listed in Appendix Table B.4.

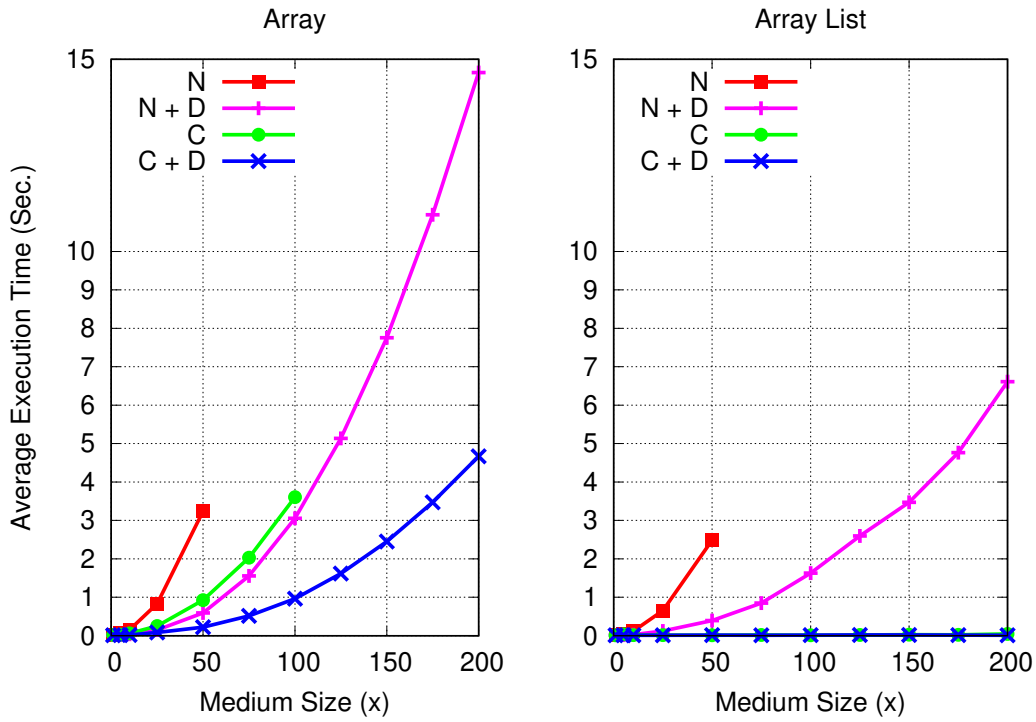


Figure 8.6: Execution time graph of lZ77 decompression on medium problem sizes

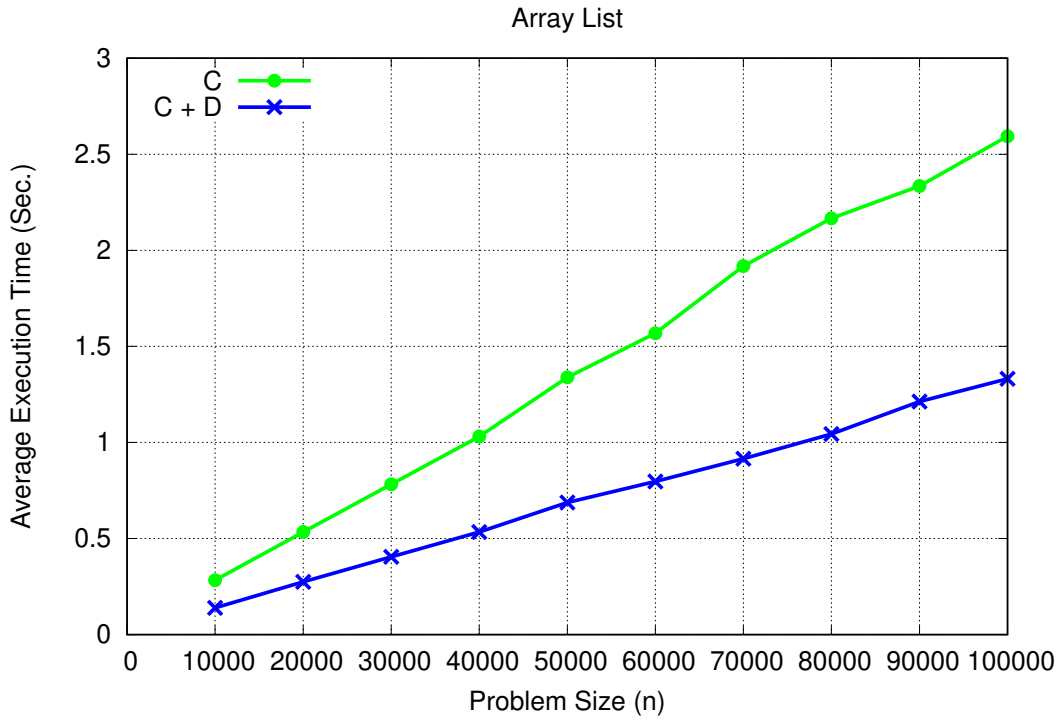


Figure 8.7: Execution time graph of LZ77 decompression using array list on large problem sizes

Figure 8.6 shows, in the case of *array*, our combined optimised code has the fastest execution time. And the naive and copy-eliminated only code fails to run on large problem sizes, due to severe memory leaks. And de-allocation only code has the slowest execution. All these four kinds of code runs in quadratic time $O(n^2)$. *array list* improves the speeds of LZ77 decompression, particularly copy eliminated and combined optimised code.

Execution Time on Large Files We increase the file sizes to eliminate measure errors of execution time and to investigate the time complexity of array list and copy elimination. To generate large compressed files, we run LZ77 compression program across a variety of large input files and write out compressed data to output files, ranging from 15.3 to 153 MB. The detailed benchmark results are listed in Appendix Table B.5.

Figure 8.7 shows, using array list lets copy eliminated (C) and combined optimised (C+D) code both have a linear time complexity. But the copy eliminated only code runs slower than the combined optimised code by an order of two magnitudes.

8.4.3 Handwritten Code and Performance

LZ77 test case has two part: compression and decompression. We convert these LZ77 Whiley programs into C code manually and benchmark these written code on the same standalone machine.

8.4.3.1 Handwritten LZ77 compression

The LZ77 compression program using preallocate array is translated from Whiley to C code by hand (see Appendix B.12). Similar to our optimised code, the handwritten C code removes unneeded array copies and also includes `free()` to avoid memory leaks.

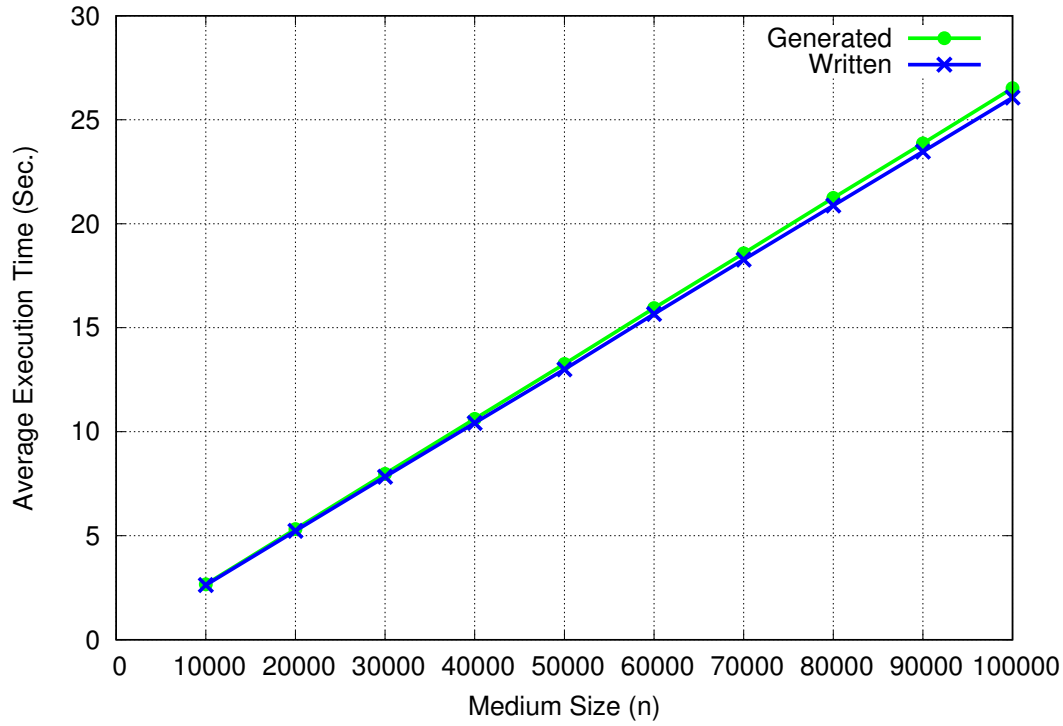


Figure 8.8: Execution time graph of written LZ77 compression code

We use the execution time of written code as base-line to compare that of our generated code and the slow-down is defined as below:

$$\text{Slow-down} = \frac{T_g - T_w}{T_w}$$

where T_g is the average time of generated code and T_w is the average time of written code. The detailed benchmark results are listed in Appendix Table B.6.

Figure 8.8 shows our generate code runs slightly slower ($1.32\% \sim 1.98\%$) than the handwritten code and the slow-downs do not increase with problem sizes. Both the written and generated code can scale to larger sizes and the time complexity is linear to problem size $O(n)$, which is the same as our optimised code.

8.4.3.2 Handwritten LZ77 Decompression

The LZ77 decompression program using array list is translated to C code manually (see Appendix B.13). We remove the unneeded overheads of array copying and reduce the memory usage in the written code. Then we benchmark

the written code with a variety of problem sizes to measure the slow-down of generated code. The detailed benchmark results are listed in Appendix Table B.7.

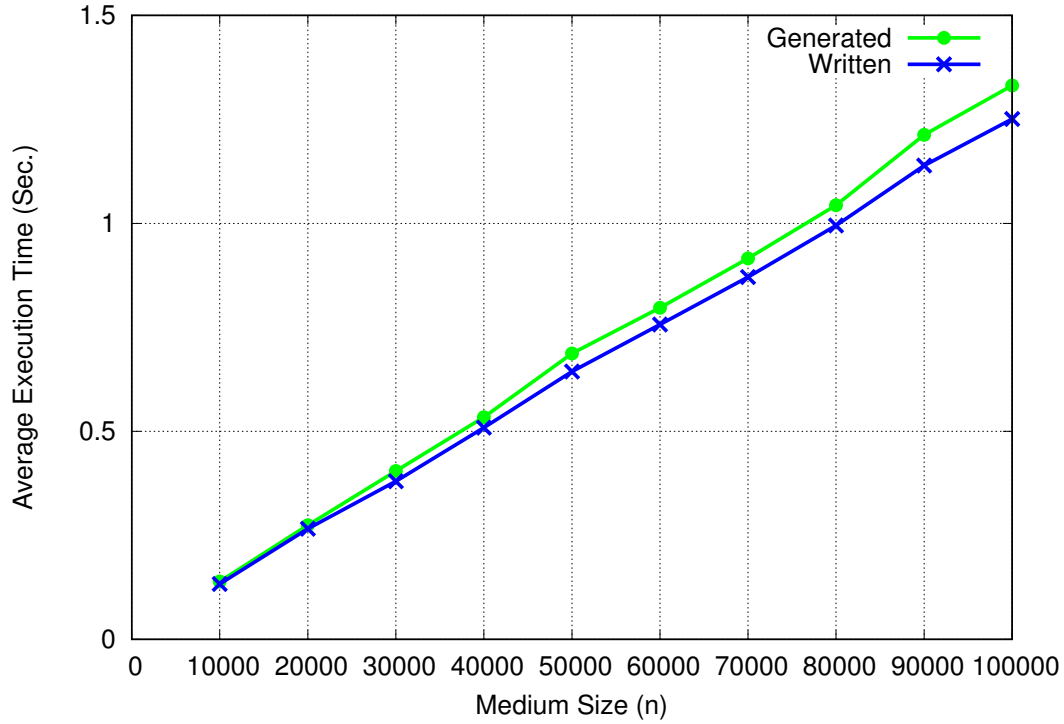


Figure 8.9: Execution time graph of generated and written LZ77 decompression code

Figure 8.9 show our generated code runs slight slower (3.22% $\sim$ 6.67%) than hand written code. But the generated and handwritten code both have similar program scalability and time complexity.

In conclusion, our generated code runs slightly slower (1.3% $\sim$ 6.6%) than the written code in both compression and decompression stages. Despite the subtle difference on speeds, our automatic generated code can maintain similar efficiency as handwritten code.

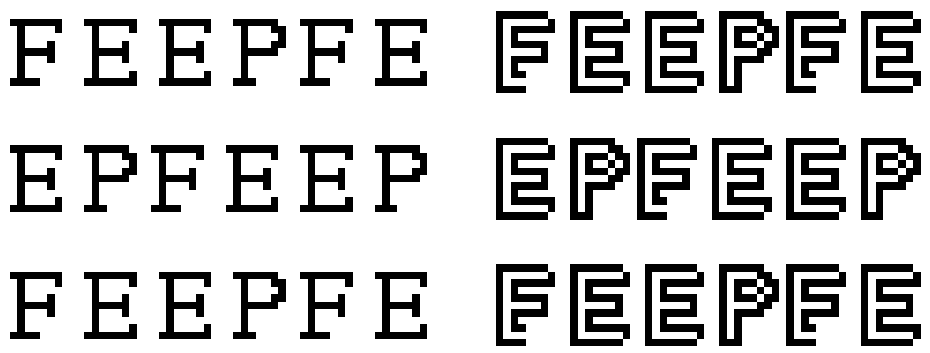
8.4.4 Conclusions

LZ77 benchmarks has three interesting results. First, in both compression and decompression programs, our copy elimination and deallocation analysis can effectively minimise the overheads of array copies and avoid all the memory

leaks to achieve a better program efficiency. Second, we find that the array append operation in the program can be replaced with pre-allocated array or array list to further reduce time complexity from quadratic $O(n^2)$ down to linear $O(n)$ or logarithmic $O(log_2 n)$, and improve the overall performance. Third, our generated code has the same amount of array copies as hand-written with $1\% \sim 6\%$ performance loss.

8.5 Case Study: Sobel Edge Detection

Our Sobel operator (Sobel, 1990) takes a black-and-white image as input, detects edge pixels and produces an image with emphasising edges as output. The algorithm computes and approximates the gradient for each pixel using convolution operator with kernels, and then compares the gradient value against the given thresholds to decide whether the pixel is an edge, and outputs the results as a byte array.



(a) Input Image (b) Output Image

Figure 8.10: Sample images before and after Sobel edge detection

The input and output images follow portable bit map (`PBM`) file format of `Netpbm` package, as shown in Figure 8.10. PBM format describes an image as a plain ASCII file with a matrix of rows and columns of pixels, and each pixel

is 0 or 1 (0:white and 1: black). And then we can convert these PBM images to different graphic formats for viewing and exchanging.

8.5.1 Algorithm

Sobel edge detection reads the input image as a single dimensional array *pixels*. The pixel value at position $p(x, y)$ can be obtained by using $pixels[x+(width*y)]$ formula. Then Sobel operator uses mathematical convolution, denoted by '$*$', to approximate the gradient G of each pixel in input image.

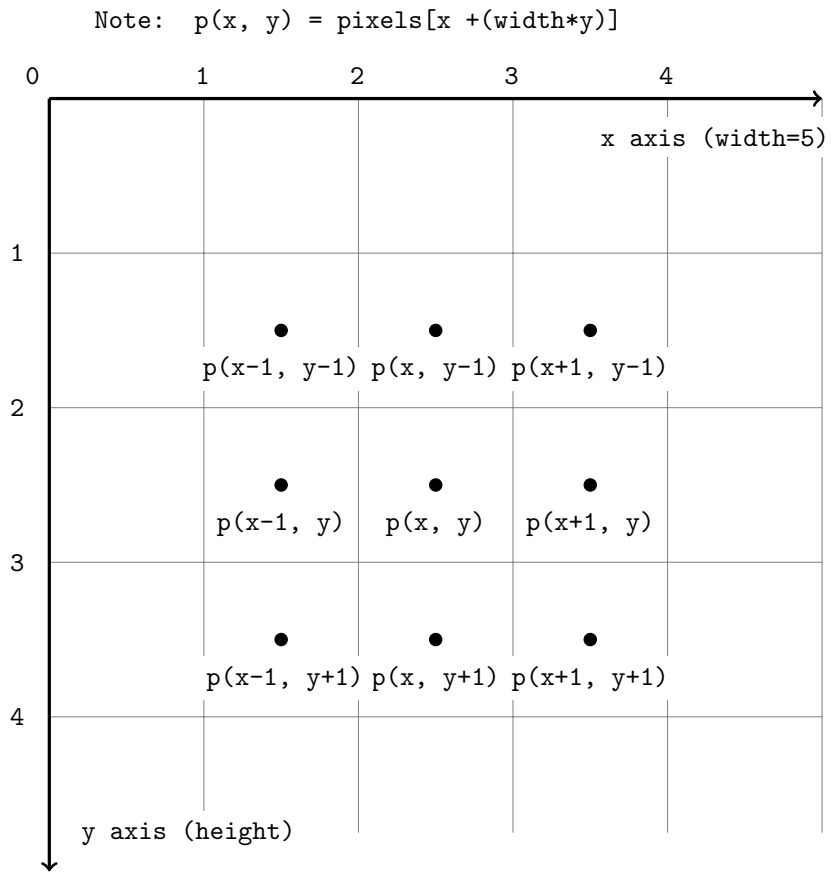


Figure 8.11: Pixel point and its neighbouring points

The convolution operator $*$ takes one point $p(x, y)$ and its neighbouring 8 pixels, as shown in Figure 8.11, to compute vertical gradient $G_{p,v}$ and horizontal gradient $G_{p,h}$ with 3x3 kernel v and h respectively, as shown in below equation 8.3 and equation 8.4. By doing so, we can intensify the edge pixel in

both vertical and horizontal direction, and make it more detectable.

$$
\begin{aligned}
G_{p,v} &= v * p(x,y) \\
&= \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} * \begin{bmatrix} p(x-1,\ y-1] & p(x,\ y-1) & p(x+1,\ y-1) \\ p(x-1,\ y) & p(x,\ y) & p(x+1,\ y) \\ p(x-1,\ y+1) & p(x,\ y+1) & p(x+1,\ y+1) \end{bmatrix} \\
&= \sum_{j=0}^{2} \sum_{i=0}^{2} v[i,\ j) \times p((x+i-1),\ (y+j-1))
\end{aligned}
\tag{8.3}
$$

$$
\begin{aligned}
G_{p,h} &= h * p(x,y) \\
&= \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * \begin{bmatrix} p(x-1,\ y-1) & p(x,\ y-1) & p(x+1,\ y-1) \\ p(x-1,\ y) & p(x,\ y) & p(x+1,\ y) \\ p(x-1,\ y+1) & p(x,\ y+1) & p[x+1,\ y+1) \end{bmatrix} \\
&= \sum_{j=0}^{2} \sum_{i=0}^{2} h[i,\ j) \times p[(x+i-1),\ (y+j-1))
\end{aligned}
\tag{8.4}
$$

We then add $G_{p,v}$ and $G_{p,h}$ each squared to get the square of total gradient G_p, and then compare it against threshold value TH squared to decide if a pixel is an edge, as follows.

$$G_p^2 = G_{p,v}^2 + G_{p,h}^2 > TH^2 \qquad \text{if 'p' pixel is an edge}$$

By comparing the total gradient against threshold, we can distinct edges of input images and then colour the edge pixel as black.

(a) Threshold = 500 (b) Threshold = 800 (c) Threshold = 1100

Figure 8.12: Sobel edge detection with varying threshold values

Figure 8.12 shows the output of edge detected images with three different thresholds. Lowering threshold value yields stronger edges because it brings in noisy and non-existing edges to output. But heightening threshold produces blurred edges because it loses some exiting edges.

In our benchmark, we choose 800 as a proper threshold value because it has the most edges from input images.

```
constant TH is 640000 // Threshold value (800*800) controls edge number
// Compute convolution on pixels[xCenter, yCenter]
function convolution(byte[] pixels, int width, int height, int
    xCenter, int yCenter, int[] kernel) ->int:
  int sum = 0
  int kernelSize = 3
  int kernelHalf = 1
  int j = 0
  while j < kernelSize:
    int y=Math.abs((yCenter+j-kernelHalf)%height)
    int i = 0
    while i < kernelSize:
      int x=Math.abs((xCenter + i - kernelHalf)%width)
      int pixel = Byte.toInt(pixels[y*width+x])// pixels[x, y]
      int kernelVal = kernel[j*kernelSize+i]  // Get kernel[i, j]
      sum = sum + pixel * kernelVal//sum += pixels[x, y]*kernel[i, j]
      i = i + 1
    j = j + 1
  return sum// 'sum' : convoluted value at pixels[xCenter, yCenter]

// Perform Sobel edge detection
function sobelEdgeDetection(byte[] pixels, int width, int height) ->
    byte[]:
  int size = width * height
  // The output image of sobel edge detection
  byte[] newPixels = [SPACE;size] // A blank picture
  // vertical and horizontal sobel filter (3x3 kernel)
  int[] v_sobel = [-1,0,1,-2,0,2,-1,0,1]
  int[] h_sobel = [1,2,1,0,0,0,-1,-2,-1]
  // Perform sobel edge detection
  int x = 0
  while x<width:
    int y = 0
    while y<height:
      int pos = y*width + x
      // Get vertical gradient
      int v_g = convolution(pixels, width, height, x, y, v_sobel)
      // Get horizontal gradient
      int h_g = convolution(pixels, width, height, x, y, h_sobel)
      // Get total gradient using absolute value
      int t_g = v_g*v_g + h_g*h_g
      // Large threshold value generates few edges
      if t_g > TH:
        newPixels[pos] = BLACK// Color pixel as black
      y = y + 1
    x = x + 1
  // All done
  return newPixels
```

Listing 8.11: Sobel Edge Whiley Program

List 8.11 shows Sobel edge detection Whiley program (full version sees Appendix B.11) with threshold value of 800. Function *sobelEdgeDetection* performs Sobel operator to estimate the total gradients for all pixels in input image, filter out non-edges with thresholds and colour the edges on output image. Function *convolution* convolutes the given pixel $p[xCenter,\ yCenter]$ with passed kernel (vertical or horizontal one), and returns the resulting gradient value.

8.5.2 Benchmark Results

Table 8.10: Memory leaks (bytes) of Sobel edge detection

	Implementation			
Problem Size	N	N + D	C	C + D
image64x64 (4.2 kB)	34,171,344	0	10,552	0
image64x128(8.3 kB)	135,449,072	0	18,776	0
image64x256 (16.6 kB)	539,331,056	0	35,160	0

Memory Leaks Table 8.10 shows our de-allocation analysis can effectively avoid all memory leaks both in de-allocated and combined optimised code. If the leaks are measured by size of image pixels n, then the naive (N) code has closely quadratic space complexity $O(n^2)$ and copy eliminated (C) code has $O(n)$ space complexity.

Each pixel is represented by 1 byte integer (`uint8_t`) and thus, the leaks of copy eliminated code shows our copy elimination avoids all unnecessary array copies and keeps only two arrays for each image. For example, the input array of `image64x256 (width=64 height=256)` amounts to 16,384 (64×256) bytes, and the total number of leaks at copy eliminated code is roughly two times of input array with some extra constant memory waste (2.3 KB).

Execution Time on Small Images We increase the height of `image64x64` by 1 to 10 magnitudes and produce input images of our small benchmarks. The detailed benchmark results are listed in Appendix Table B.8.

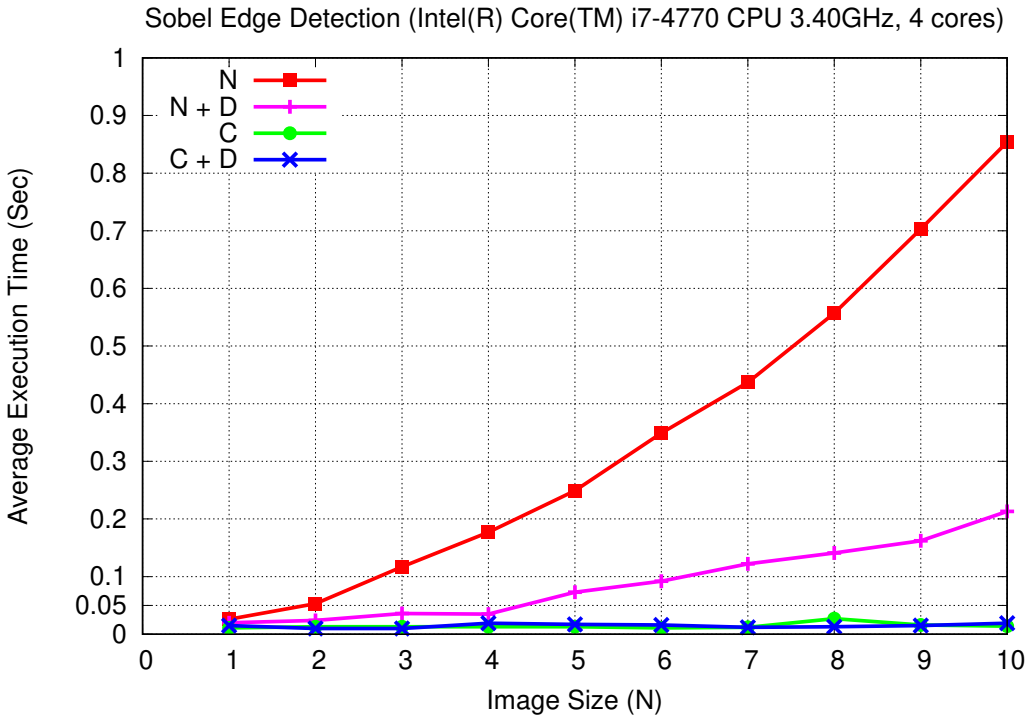


Figure 8.13: Execution time graph of Sobel edge on small problem sizes

Figure 8.13 shows that, as the problem size increases, our copy eliminated (C) and its combined deallocation (C+D) code both have the fastest execution. De-allocated only (N+D) code runs slightly slower than copy eliminated code, due to expensive overheads imposed by array copies, but still outperforms the naive code. And the naive (N) code has the slowest $O(n^2)$ quadratic time complexity if measured by input file size n.

The naive (N) code grows non-linearly with problem size increases, and has longer latency on large files. De-allocated only (N+D) code runs at faster speeds than naive code and has roughly $O(n)$ linear time complexity if measured by problem size. Copy eliminated (C) and combined optimised (C+D) code both are the fastest execution, but we can not see the time variation from the graph, due to their small running time.

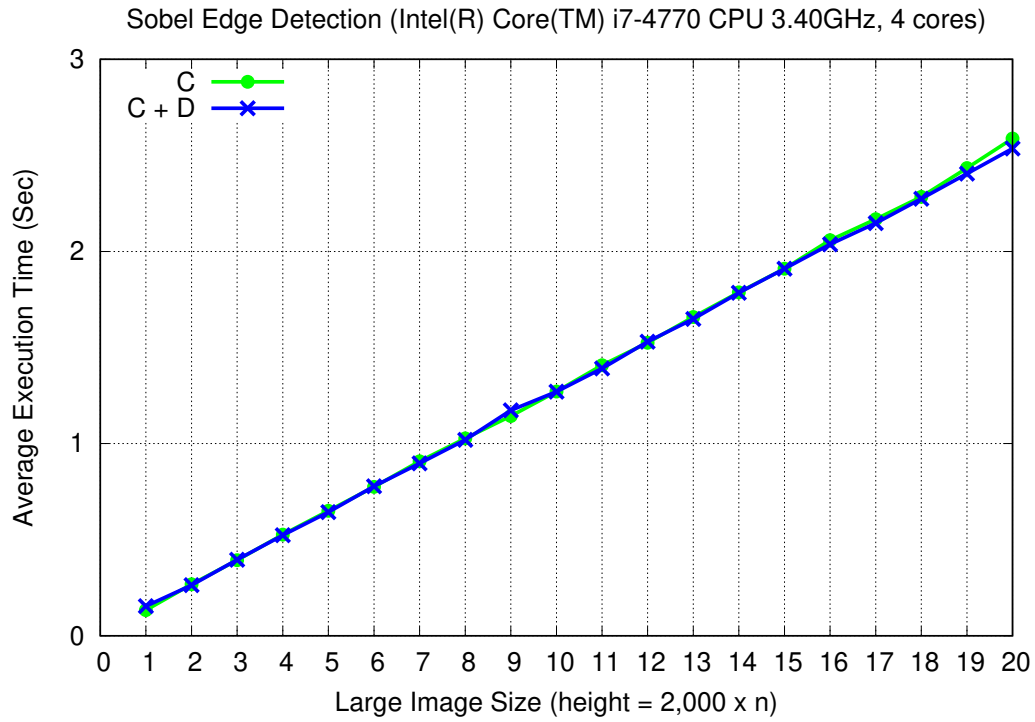


Figure 8.14: Execution time graph of Sobel edge on large problem sizes

Execution Time on Large Images We therefore conduct a large benchmark and measure the execution time of copy eliminated and combined optimised code. We take `image2000x2000` (width=2,000 height=2,000) as base image size, and multiply the height by 1 to 20 times and fill in each item with a byte number, ranging from 0 to 255, and produce input images for large benchmarks. The detailed benchmark results are listed in Appendix Table B.9.

Figure 8.14 shows our combined optimised code (C+D) runs as fast as copy eliminated (C) code, and also shows our extra de-allocation efforts in this test case does not significantly affect the performance nor slow down the execution. The copy optimised Sobel edge program with/without deallocation has linear time complexity $O(n)$ with problem size.

8.5.3 Handwritten Code and Performance

Our project takes a Whiley program as input and produces efficient C code by our automatic code generator. One may be interested in the performance of

handwritten C code. In this section, we take Sobel edge detection as a test case to manually translate the Whiley program into C code (see Appendix B.14), and then run the benchmarks to compare the performance with our automatic generated code. The written code uses only two copies of arrays to hold input and output image pixels, and includes two `free()` statements to release the allocated memory of these two arrays.

We use 64-bit (`int64_t`) integers as default type on both generated and handwritten code. After analysing the source program, we notice that Sobel edge detection is a computation intensive application and requires lots of integer arithmetic. That is, for each image pixel the Sobel operator weights its value with the kernel matrix by applying the convolution operation of the 3x3 matrix multiplication and summation.

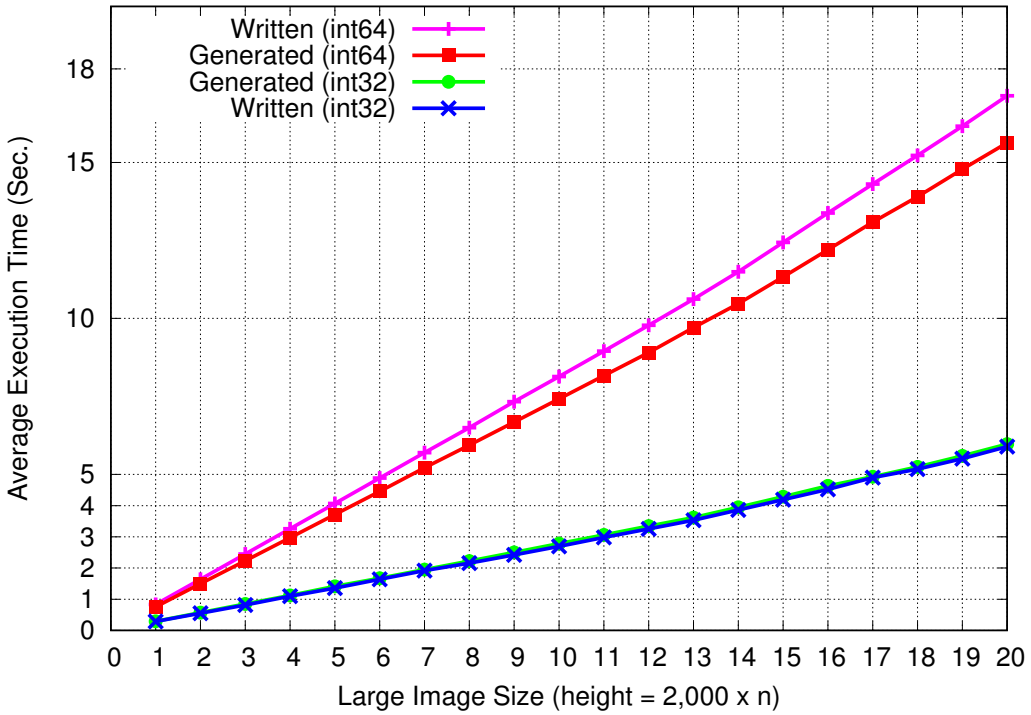


Figure 8.15: Execution time graph of written Sobel edge code at `O2` optimisation

To investigate whether the use of integer types affects the performance, we experiment the generated and handwritten with both 64-bit (`int64_t`) and 32 integer (`int32_t`) types. Also, we experiment two kinds of GCC compiler

optimisations: level 2 (`O2`) and level 3 (`O3`). Level 3 turns on all optimisations specified by level 2 and also enables more optimisation options, e.g. loop vectorisation transforms the loop and improve the performance of resulting code at the expense of longer compilation time and increasing debugging efforts.

Level 2 Compiler Optimisations We compile the generated and handwritten code with `-O2` optimisation level and run the program across a variety of problem sizes. Figure 8.15 shows that, 32-bit integers can make the generated and written Sobel edge program run faster than 64-bit integers, and the speed-ups however stays flat at a factor of 2.6x and 3.0x in generated and written code respectively and do not grow with the problem sizes.

Level 2 compiler optimisation makes the handwritten code run slightly faster (1.03x) than generated code with 32-bit integers but run slower with 64-bit integers. We will try a more aggressive compiler optimisation to speed up the Sobel edge program.

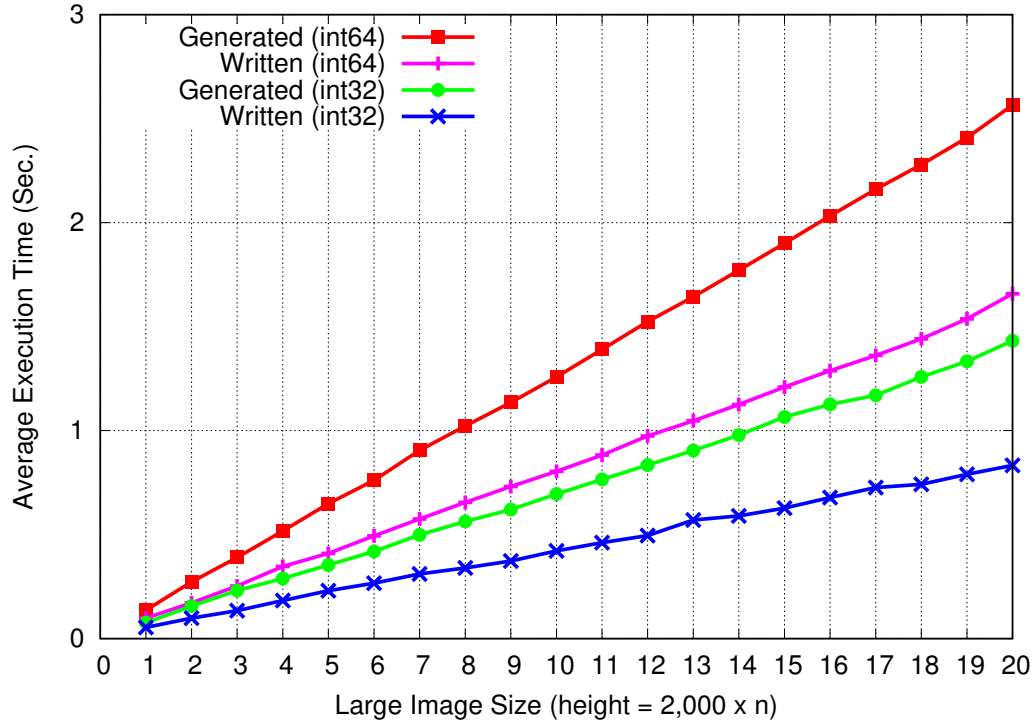


Figure 8.16: Execution time graph of written Sobel edge code at `O3` optimisation

Level 3 Compiler Optimisation We apply `-O3` optimisation on both handwritten and generated code to gain further performance improvement and then run the benchmarks again. The detailed benchmark results are listed in Appendix Table B.10.

Figure 8.16 shows that that `-O3` optimisation improves the overall performance of generated and written Sobel edgee programs more than level 2 such that the execution time is reduced from 16 down to 0.8 seconds. Similar to results of level 2 optimisation, using 32-bit integers achieves a better speedup than using 64-bit type (1.8x and 1.9x in generated and handwritten code respectively). And for the same type of integers, the generated code runs at least 50% slower than handwritten code. Therefore, the fastest execution is the 32-bit integer version of handwritten code, followed by 32-bit generated code and 64-bit handwritten code. And the slowest is 64-bit generated code.

The 32-bit integer types runs more efficiently than 64-bit integers in both generated and written code, and the generated code is slower than handwritten code. In this graph, we can see that the generated and handwritten code both have linear time complexity $O(n)$ and scale to larger problem sizes.

Summary The benchmark results are summarised as follows. First, Sobel edge detection heavily relies on the integer arithmetic and thus its performance can be affected by the choice of integer types. On our standalone machine, 32-bit integer type (`int32_t`) provides a better efficiency than 64-bit type (`int64_t`) as it takes up half of space in memory and less time to perform integer arithmetic. Therefore, using 32-bit integers makes Sobel edge operation fast.

Second, handwritten code obtains more performance gain from GCC level 3 optimisation than generated code, because the compiler can fully optimise the handwritten code and gain a substantial speed-up in the running time. Let us consider the most expensive function *convolution* of Sobel edge program. The operator multiplies 3x3 matrices and sums up the total by using the following

loop nest.

```
int j = 0
while j<3:
  ...
  int i =0
  while i<3:
    ...
    sum += pixel[x*width+y] * kernel[j*3+i]
    i = i + 1
  j = j + 1
```

The inner and outer loops both are known to iterate 3 times and no dependency exists between *pixel* and *kernel* matrices. Then, GCC compiler detects such an loop nest can be optimised and unrolls the inner and outer loops into sequences of operations, shown below.

```
...
sum += pixel[x*width+y] * kernel[0]
sum += pixel[x*width+y] * kernel[1]
sum += pixel[x*width+y] * kernel[2]
...
sum += pixel[x*width+y] * kernel[8]
```

The loop unrolling pre-calculates the array index and thus reduces the number of arithmetic operations at run-time. Also, we take out loop conditions and do not generate conditional jumps in the machine code so that branch penalty can be avoided and the program speed can be increased.

We compile our generated and handwritten code into assembly code at level 3 optimisation. We observe that in handwritten code, GCC compiler can fully understand the loop nest and unroll both inner and outer loops to produce better optimised executable and gain speed-ups. However, in our generated code GCC compiler transforms only the inner loop into a sequence of instructions but keeps the outer loop as it is, because our generated code includes a number of temporary variables which do not appear in the handwritten code, and makes the program analysis too complicated to carrying out a full loop optimisation.

Due to extra temporary variables, our generated code has less loop unrolling optimisation enabled at level 3 of GCC compiler, and thus runs $60\% \sim 70\%$ slower than handwritten code.

8.5.4 Conclusions

Sobel edge detection benchmark has three interesting results. First, our copy elimination analysis can reduce the array copying overheads of our naive code from quadratic time complexity $O(n^2)$ to linear $O(n)$, and then combines with our de-allocation analysis to produce an efficient and memory leak-free code. Second, our implementation runs 52% slower than 64-bit integer version of handwritten code. Third, we also find the performance of the Sobel edge program can be improved further using bound analysis to automatically produce code with 32-bit integers.

Chapter 9

Benchmarks for Parallel Programs

Parallel computing is heavily used in data analytics to speed up vast amounts of data processing and produce the results timely. In particular, the in-memory parallel/distributed computing gains more focus for its low latency and high scalability. Most importantly, almost all modern laptops or desktops have already multi-core CPUs.

In Chapter 8, our benchmark results show our compiler can produce good and fast sequential code for most of the cases. However, our memory optimisation does not have significant performance improvement on `BubbleSort` and `MatrixMult` cases. By profiling the generated C code, we notice that these two programs are CPU-bound applications as their computation dominates the entire execution time and results in performance bottleneck. So more computing resources, instead of reducing memory overheads, are needed to make these programs run faster.

We conducted a feasibility study to evaluate the difficulties of a parallelising compiler that can transform a sequential program into the parallel code using analysis techniques, and to know whether the parallel code can gain further speed-ups from concurrent computing. We explore several case studies and conduct parallel experiments on standalone computers as well as virtual

machines on several cloud platforms. Unfortunately, our benchmark results are disappointing, and only two cases exhibit scalable and useful speed-ups with the number of threads.

The parallelising compiler is not implemented in our Whiley-to-C project because its difficulties exceed our expectations, and we lack time to accomplish it. So in this chapter we present a number of hand-on experiences to transform the sequential C code, produced and optimised by our back-end, into parallel applications with `Polly` automatic compiler, or manually rewrite the C code with `OpenMP` and `Cilk Plus` libraries to take up parallel opportunities with their runtime environments. Benchmark results are also included to show the effectiveness of each parallellising approach.

This chapter is structured as follows. Section 9.1 gives an introduction of OpenMP work-sharing parallel model. Section 9.2 benchmarks Polly automatic compiler on our micro benchmark programs. Section 9.3 exploits the task parallelism of *MergeSort* program with Cilk Plus parallel expressions. Section 9.4 parallelises *CoinGame* C code with OpenMP, Cilk Plus, and Polly compiler, and then compare the performance of these three parallel techniques. Section 9.5 uses map-reduce programming model to parallelise *LZ77compression* case studies. Through these practices we hope to provide a way of how to alter our compiler to generate parallel code.

9.1 OpenMP Data/Task Parallelism

`OpenMP` (Chapman et al., 2008) provides API for programmers to write shared memory parallel programs in C/C++ and Fortran languages. OpenMP runtime is based on fork-join model to divide the target task into a number of smaller sub-tasks and create threads to run each sub-task in parallel and then at a subsequent program point merge all the results of sub-tasks to one final result of the execution. For example, the below loop in a sequential code can be explicitly declared as OpenMP parallel part using `OpenMP pragma`, shown

below program.

```
1 int a[1000] = ...;
2 int b[1000] = ...;
3 int c[1000] = ...;
4 // Start of parallel region
5 #pragma omp parallel for
6 for(int i=0;i<1000;i++){
7   c[i] = a[i] + b[i] // Work is distributed among all threads.
8 }
9 // Implicit barrier at end of parallel region (#pragma omp barrier)
```

The compiler directive `#pragma omp parallel for` specifies the loop must be executed with multi-threads. So when entering the parallel region of loop, OpenMP work-sharing run-time creates a team of threads, splits up the entire loop iterations into a number of parts and then distributes each part of loop among the team of worker threads, as shown in the following graph:

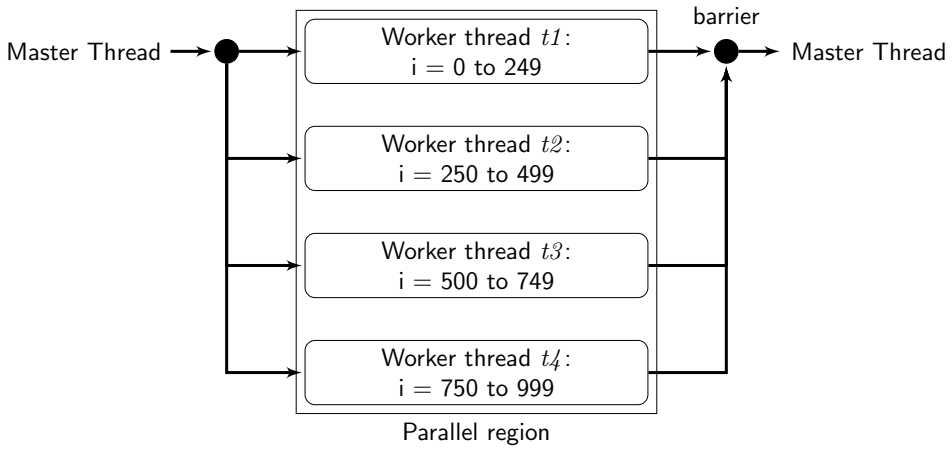


Figure 9.1: OpenMP work-sharing parallel programming model

Array a, b and c are shared among all threads within parallel region. The master thread, which runs the sequential part of the program, breaks down the loop iterations into 4 parts. Because each thread processes only one part and each part does not overlap or has any data dependency, all threads can perform their computation on the shared arrays in parallel without causing any data conflicts.

Lastly the run-time implicitly adds a synchronisation barrier at the end of parallel region to keep each worker thread waiting at barrier point until

all threads are completed. Using a barrier can guarantee the master thread does not use any unfinished data after the parallel region and produces wrong results.

There are advantages and disadvantages about OpenMP. First, OpenMP parallel programming language model can be carried out across heterogeneous multi-threaded machines, but a compiler that supports OpenMP is required to compile OpenMP programs into parallel execution. A number of compilers have a built-in implementation for OpenMP, including `GCC`, `LLVM Clang` and `Intel` C/C++ compiler.

Second, OpenMP parallel program looks alike to the sequential one with additional compiler directives and allows programmers to experiment different kinds of parallelism, such as map-reduce parallel model (see Section 9.5.2) and to gain further speed-ups. However, extra care for synchronisation is required to avoid race conditions and increases debugging difficulty.

Third, OpenMP run-time automatically decomposes the tasks and makes a load-balancing schedule to run all threads efficiently. However, some OpenMP programs have lower parallel efficiency and do not scale up to the processor number, because

- The program has a large portion of sequential execution, so leads to a small part of code parallelised. According to Amdahl's Law, the speed-up is determined by the fraction of parallel computation and the number of processors:

  $$Speedup = \frac{1}{(1-f)+(f/p)} \simeq \frac{1}{1-f} \quad \text{when p is } \infty$$

  where f is the parallel percentage in a program and p is the number of processors. When we increase the processor number p to extremely large, we have $\frac{f}{p}$ so small and close to zero that we can omit it in the speed-up calculation.

  Therefore, regardless of how powerful cores the machine hardware has, the maximal speed-up is limited by the parallelism coverage, which is the

percentage of computation that runs in parallel, so we need to exploit as much parallelism as possible in the program to increase the portion of parallel OpenMP code and gain more speed-ups.

- Barrier synchronisation protects the shared data in OpenMP programs but may introduce potential false-sharing problems. Let us consider our example again. The same array c is updated by four threads, and because array c is stored in the same shared memory address, each update will force the entire memory stall and keep other three threads waiting until the update operation finishes.

  The false sharing not only degrades the performance of OpenMP parallel execution but results in poor scalability. We may eliminate false-sharing by padding the arrays so that each array element is in different and distinct memory address/cache line, and thus each update can be operated independently and concurrently without waiting overheads.

Lastly, OpenMP provides loop-level parallelism to decompose the loop iterations among all threads to distribute the computation in parallel. Also, OpenMP can use divide-and-conquer parallelism technique to continuously split a task into small sub-tasks until each sub-task has a relatively fine-grain to be executed directly on the single processor. We will illustrate these two types of parallelism with the coin game and LZ77 compression test cases.

9.2 Polly Compiler Data Parallelism

LLVM (Low Level Virtual Machine) (Lattner, 2008) is a target and programming language independent code representation, and allows a variety of compiler optimisation and code generator to produce efficient LLVM code that runs efficiently on different hardware.

The Polly (Polyhedral Optimisation for LLVM) (Grosser et al., 2012) provides automatic code transformation for LLVM code and produces platform-independent optimised sequential and parallel code.

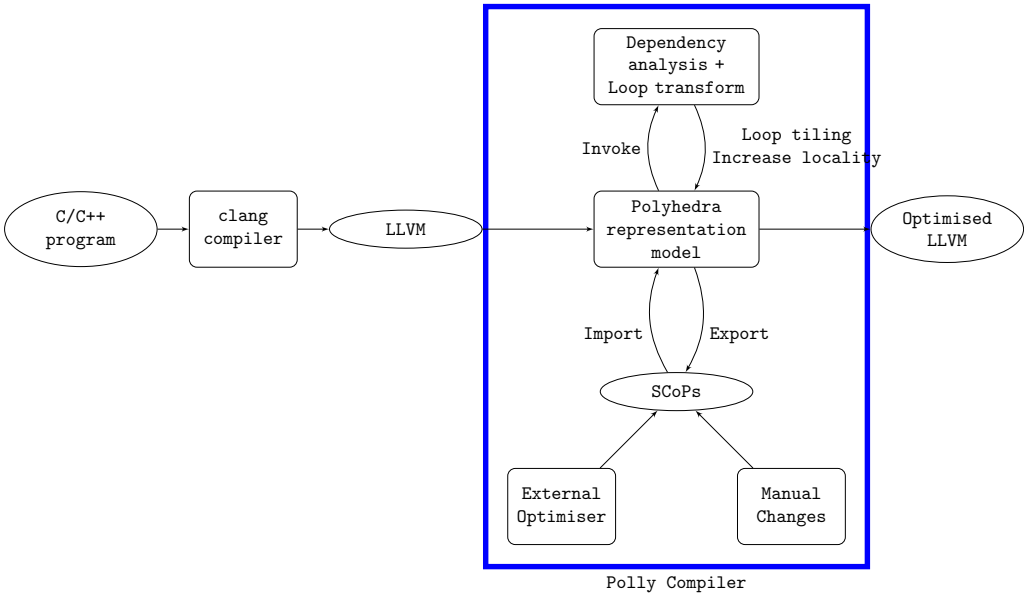


Figure 9.2: Polly architecture

9.2.1 Polly Compiler

Polly compiler (Grosser et al., 2012) (see Figure 9.2) takes as input LLVM code, translated from a C program by `clang` compiler, and then analyses loop kernels and produce optimised LLVM code as output. Polly compiler uses polyhedral techniques to optimise the data locality and parallelism in LLVM. It detects parallelisable code sections in LLVM and translates them into polyhedral description or static control parts (SCoPs). Then the polyhedral optimiser is enabled to analyse SCoPs and apply the optimisation, i.e. changing execution order of statements in a SCoP and the memory access of a SCoP. After SCoP transformation, Polly (Raghesh, 2011) can translate the detected parallel loops into OpenMP code and replace SIMD instructions with parallel execution.

Polly project has been actively improved since its first creation. The parallel reduction technique, such as concurrent sum operator, was introduced in Polly polyhedral optimiser (Doerfert et al., 2015) to identify possible parallelism of data-dependent loops and generate more efficient scheduling. Multi-dimensional variable array access (Grosser et al., 2015a) enabled in Polly makes

the polynomial array problem solvable to a linear solver. AST (Abstract Syntax Tree) generator with the support of Presburger arithmetic (Grosser et al., 2015b) was implemented in Polly to enable the validation of user-specific optimisation by translating polyhedra programs into an AST, walking through that AST and checking constraint conditions.

Polly is being used in high performance applications. KenelGen (Mikushin and Likhogrud, 2012) compiler used Polly LLVM Polyhedra analysis to automatically transform while-loops to parallel for-loops, and to port the code running on GPUs. Polly was also used to speed up the Lattice Quantum Chromodynamics (QCD) program (Kruse, 2014) running on an IBM Blue Gene/Q supercomputer, as its optimisation on statement execution orders and data clusters not only improves the data flow, but reduces transfer overheads across the distributed system.

A new polyhedral model (Moll et al., 2016) is proposed in Polly to automatically split input data space and produce less-divergent OpenCL kernel, so that each kernel would access the memory space concurrently without barriers and generate fewer numbers of instructions to utilise the parallel computing powers on GPUs.

9.2.1.1 Static Control Parts (SCoPs)

Polly optimiser transforms the iteration space of a loop into smaller blocks, so that each block fits into the cache size of CPU. The data required in each block stays in the same CPU cache line, so data locality can be increased to achieve better performance. The loop tiling includes:

- **Interchange** changes the execution order of inner and outer loops,
- **Fission** splits one nested loop into two independent loops.
- **Strip mining** transforms a single loop into a nested loop with a strip.
- **Unroll-And-Jam** unrolls most of the loops, except for the innermost one.

- **Loop blocking** combines 'interchange' and 'strip mining' to increase the data locality.

Polly compiler can be used to expose the parallelism of our generated C code and produce parallel OpenMP code, which can be run with multiple threads. The parallel loop is qualified and converted into Polyhedral model and represented as a SCoP. Polly compiler uses region-based approach to go through each block in a control flow graph and checks if the block meets below criteria to form a valid SCoP.

- The block contains regular for-loop structure and the memory base address must be distinct or invariant. For example, data structure needs to be replaced with one dimensional array to avoid indirect memory access.

- The block contains an affine loop bound which increases linearly with loop variable.

- The block does not have any side effect.

These rules are illustrated with matrix multiplication example.

```
1  function mat_mult(int[] a, int[] b, int[] c, int width, int height) ->
       (int[] c):
2    int i = 0
3    while i < height:
4      int j = 0
5      while j < width:
6        int k = 0
7        while k < width:
8          // c[i][j] = c[i][j] + a[i][k] * b[k][j]
9          c[i*width+j] = c[i*width+j] + a[i*width+k]*b[k*width+j]
10         k = k + 1
11       j = j + 1
12     i = i + 1
13   return c
```

Listing 9.1: Original matrix multiplication program

Example 9.1 *Consider the nested loops in matrix multiplication program (see Listing 9.1). Function mat_mult takes two arrays a and b as input, and multiplies $a[i][k]$ by $b[k][j]$ and sums up the total to produce the entry $c[i][j]$ in output array c. The loop tiling optimisation is described as follows.*

First, function *mat_mult* stores each matrix with one dimensional array, instead of two-dimensional arrays, as the former has steady and predictable behaviour whereas the latter may use indirect pointers and stop from being parallelised. So $c[i][j]$ is equivalent to $c[i * width + j]$.

Second, loop bounds can be calculated as affine results to make Polly compiler easily optimise and parallelise the loop. Suppose we introduce variables $i0,\ i1,\ j0,\ j1,\ k0,\ k1$ to represent the inner and outer loop variables respectively, and each of their values increases with the number of loop iteration. Then we can discover below affine expressions for all loop variables $i,\ j,\ k$:

```
1  // 1st level tiling - Tiles
2  for(int i0=0;i0<=floord(height-1, 32);i0++)
3    for(int j0=0;j0<=floord(width-1, 32);j0++)
4      for(int k0=0;k0<=(width-1)/32;k0++) {// k0 loop
5        // 1st level tiling - Points
6        for(int i1=0;i1<=min(31, height-32*i0-1);i1++)
7          for(int j1=0;j1<=min(31, width-32*j0-1);j1++)
8            for(int k1=0;k1<=min(31, width-32*k0-1);k1++){// k1 loop
9              int i = i0*32+i1; // Affine expression for i
10             int j = j0*32+j1; // Affine expression for j
11             int k = k0*32+k1; // Affine expression for k
12             // Compute matrix multiplication
13             c[i*width+j]=c[i*width+j] + a[i*width+k]*b[k*width+j];
14           }// Ending k1 loop
15 }// Ending k0 loop
```

Listing 9.2: Loop-tiling matrix multiplication by Polly compiler

Each loop variable is expressed with a linear function of reference variables. For example, loop variable i can be expressed as an affine expression $i = 0 + 32 \times i0 + 1 \times i1 = c0 + c1 \times i0 + c2 \times i1$ where $c0, c1, c2$ are all constants, reference variable $i0$ has the ranges from 0 to $floord(height - 1,\ 32)$, and reference variable $i1$ is between 0 to $min(31,\ height - 32 \times i0 - 1)$ and `floord` and `min` functions are used to avoid out-of-index array errors. The affine expression is linear with reference variables are $i0$ and $i1$, and useful for further Polly optimisation.

Third, Polly compiler splits large space of loop iterations into blocks of 32 tiling size, so that the data in inner loop stays at the same cache line to compute the matrix multiplication and gain better speed-ups. In addition, output array c reads and writes the data only at a specific location at each inner

loop iteration, so does not cause any side effect. The matrix multiplication program meets all three conditions and thus forms a valid SCoP, so that Polly compiler can transform the nested loops into a parallel loop and take advantage of multi-threading computing power to speed up the execution.

9.2.1.2 Polly OpenMP Parallelism

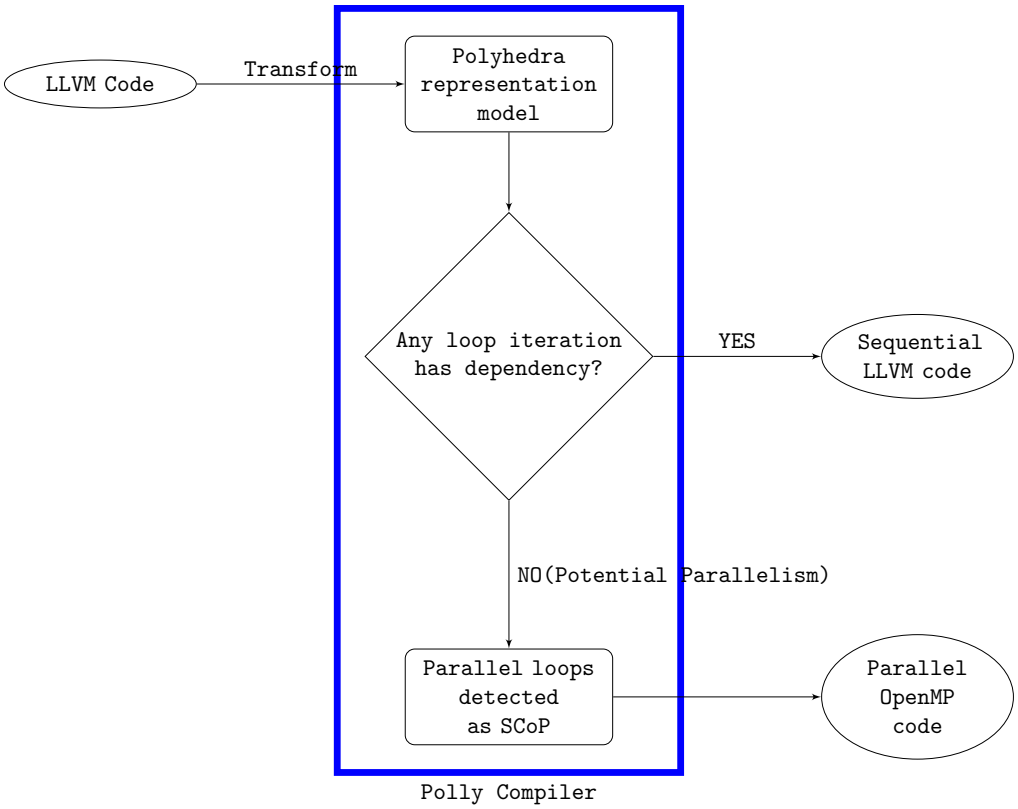


Figure 9.3: Automatic parallelisation and code generation by Polly compiler

Polly compiler(Raghesh, 2011) can automatically analyse and detect the loop parallelism (see Figure 9.3). First, the LLVM code is transformed into polyhedral representation model, to calculate data dependency. If a loop can be executed without any dependency in two executive iterations, then Polly detects and qualifies the loop as SCoPs and annotates the program part with parallel pragmas in LLVM:

- `GOMP_parallel_loop_runtime_start`
- `GOMP_parallel_end`

So the parallel loop can run in concurrently by invoking OpenMP library calls.

9.2.2 Performance Evaluation

The micro-benchmarks Whiley programs (see Section 8.1) are first translated and optimised into copy eliminated and de-allocated (C+D) code by our code generator and our analyses. Then we use Polly compiler to compile the generated C code into sequential and parallel OpenMP executable. And we use GCC compiler (v.5.4) to produce the base-line sequential executable.

The benchmark programs are run on one standalone machine (i7-4770 CPU @ 3.40GHz and 16 GB) and several kinds of cloud computing frameworks.

Table 9.1: Average execution time (seconds) of micro-benchmarks optimised by GCC and Polly compilers on standalone machine

				Polly OpenMP			
Program	Problem Size	**GCC**	**Polly Seq**	**1 thread**	**2 threads**	**3 threads**	**4 threads**
	100,000	0.0081	0.0085	0.0105	0.0144	0.0126	0.0150
Reverse	1,000,000	0.0208	0.0162	0.0635	0.0722	0.0640	0.0580
	10,000,000	0.0478	0.0416	0.2250	0.6502	0.5518	0.5054
	1,000	0.0073	0.0078	0.0077	0.0083	0.0084	0.0085
TicTacToe	10,000	0.0175	0.0158	0.0167	0.0150	0.0156	0.0152
	100,000	0.0972	0.0834	0.0896	0.0902	0.0836	0.1037
	1,000	0.0089	0.0075	0.0079	0.0075	0.0072	0.0073
BubbleSort	10,000	0.0789	0.0418	0.0782	0.0758	0.0765	0.0839
	100,000	6.6184	3.2852	6.9684	6.9509	6.9288	6.9857
	1,000	0.0063	0.0062	0.0103	0.0068	0.0065	0.0082
MergeSort	10,000	0.0083	0.0085	0.0089	0.0155	0.0110	0.0107
	100,000	0.0144	0.0167	0.0287	0.0255	0.0228	0.0252
	1,000	1.1709	0.6416	0.4704	0.2474	0.1743	0.1323
MatrixMult	2,000	15.7166	5.1205	3.7027	1.8658	1.2701	1.0275
	3,000	46.5542	17.3702	12.6093	6.2330	4.3398	3.3259

9.2.2.1 Micro-benchmark on standalone machine

We use the execution time of GCC optimised micro-benchmark programs (at `O3` optimisation level) as a baseline to evaluate the performance of Polly sequential and OpenMP code. The benchmark results of micro-benchmark programs are listed in Table 9.1.

Table 9.2: Absolute speed-ups of Polly optimised micro-benchmark programs (vs. GCC compiler) on standalone machine

Program	Problem Size	**Polly Seq**	**Polly OpenMP**			
			1 thread	**2 threads**	**3 threads**	**4 threads**
Reverse	100,000	0.96	0.78	0.57	0.64	0.54
	1,000,000	1.28	0.33	0.29	0.32	0.36
	10,000,000	1.15	0.21	0.07	0.09	0.09
TicTacToe	1,000	0.93	0.94	0.88	0.87	0.85
	10,000	1.11	1.04	1.16	1.12	1.15
	100,000	1.16	1.08	1.08	1.16	0.94
BubbleSort	1,000	1.18	1.13	1.18	1.23	1.21
	10,000	1.88	1.01	1.04	1.03	0.94
	100,000	2.01	0.95	0.95	0.96	0.95
MergeSort	1,000	1.02	0.62	0.94	0.97	0.78
	10,000	0.98	0.94	0.53	0.75	0.78
	100,000	0.86	0.50	0.57	0.63	0.57
MatrixMult	1,000	1.83	2.49	4.73	6.72	8.85
	2,000	3.07	4.24	8.42	12.37	15.30
	3,000	2.68	3.69	7.47	10.73	14.00

Table 9.2 shows that, Polly sequential code in *BubbleSort* and *MatrixMult* cases achieves at least 1.5x speedup than GCC code and a slightly good and similar performance as GCC compiler in *Reverse* and *TicTacToe* cases. But Polly sequential code has slight slow-down in *MergeSort* case.

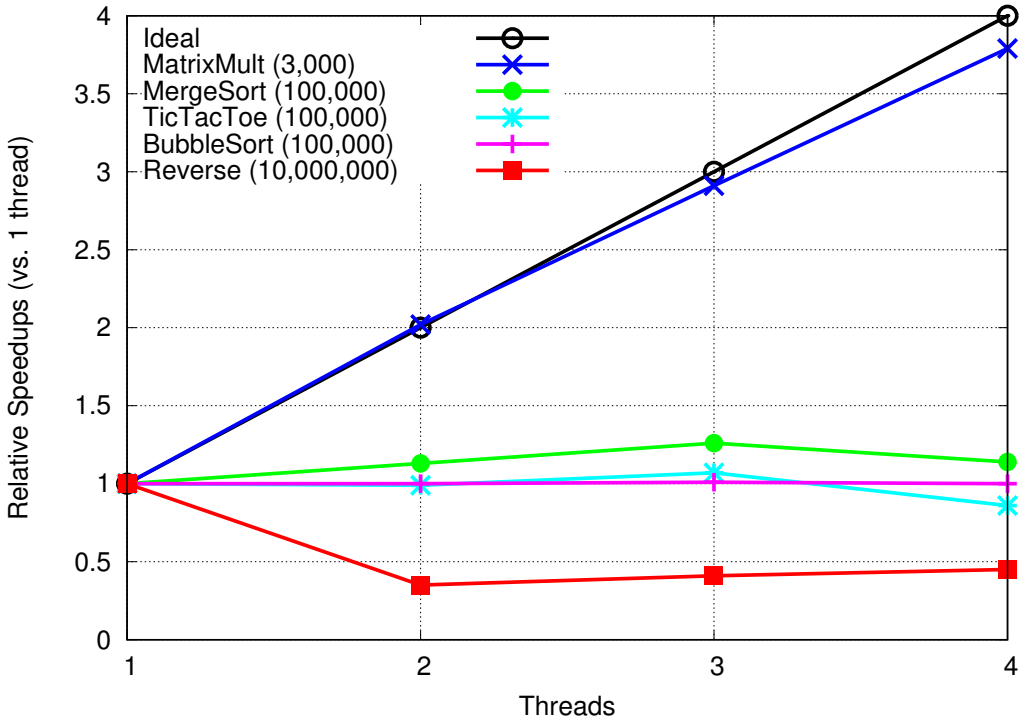


Figure 9.4: Relative speed-ups of Polly OpenMP micro-benchmark programs on standalone machine

Figure 9.4 shows relative speedups compared to the execution time of single-threaded Polly OpenMP code. Results show that Polly OpenMP code in *MatrixMult* case has excellent parallel efficiency and achieves ideal parallelism. However, Polly OpenMP code in the remaining cases does not have performance improvement and even slow-downs.

9.2.2.2 MatrixMult benchmarks on virtual machine

We use *MatrixMult* program as a test case to benchmark the performance of Polly optimisation on the below three kinds of machines:

- Standalone machine: Intel i7-4770 CPU (@ 3.40GHz, 4 cores) and 16GB
- `Amazon EC2` c4.2xlarge instance: Intel(R) Xeon(R) CPU E5-2666 v3 (@ 2.90GHz, 4 cores) and 15GB
- `Microsoft Azure` F8s standard instance: Intel(R) Xeon(R) CPU E5-2673 v3 (@ 2.40GHz, 8 cores) and 16 GB

Table 9.3: Average execution time (sec.) of *MatrixMult* case on standalone

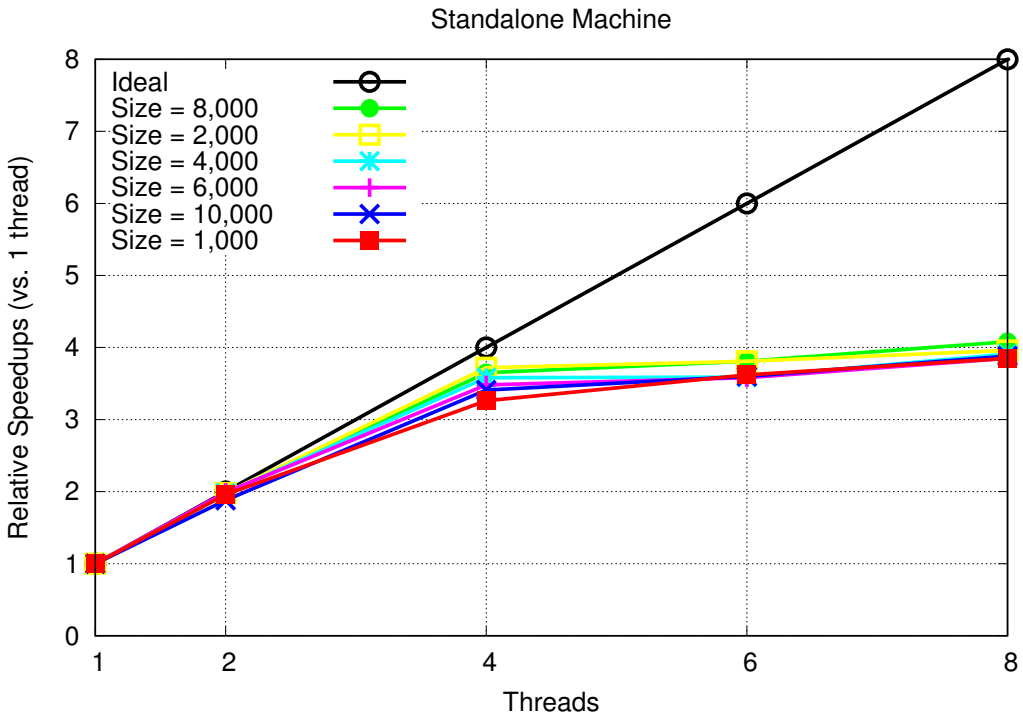
			Polly OpenMP				
Problem Size	GCC	Polly Seq	1 thread	2 threads	4 threads	6 threads	8 threads
1,000	1.4	0.6	0.6	0.3	0.2	0.2	0.2
2,000	19.0	5.0	4.6	2.3	1.2	1.2	1.2
4,000	173.9	39.8	36.8	18.6	10.3	10.3	9.4
6,000	595.9	134.0	123.5	62.0	35.5	34.5	32.1
8,000	1625.3	330.8	309.7	157.8	84.8	81.4	75.8
10,000	2636.4	622.5	573.6	304.6	168.3	159.5	147.8

Table 9.4: Average execution time (sec) of *MatrixMult* case on AWS EC2

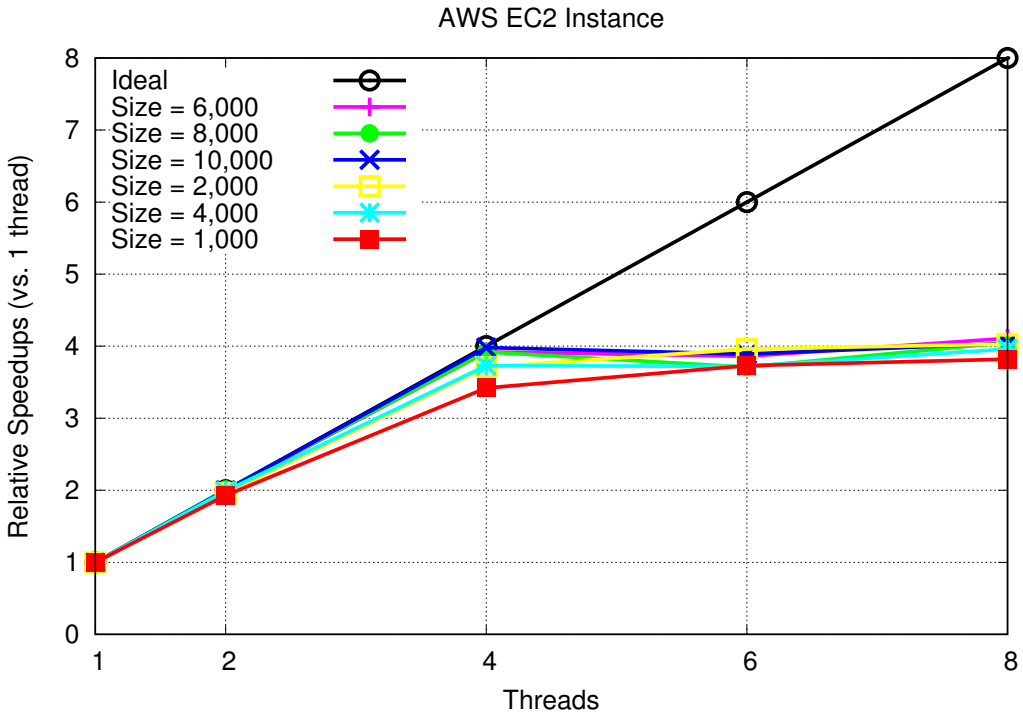
			Polly OpenMP				
Problem Size	GCC	Polly Seq	1 thread	2 threads	4 threads	6 threads	8 threads
1,000	1.2	0.7	0.7	0.4	0.2	0.2	0.2
2,000	26.6	6.1	6.0	3.1	1.6	1.5	1.5
4,000	238.8	48.8	48.3	24.3	12.9	13.0	12.2
6,000	821.6	167.2	165.1	82.6	42.1	42.8	40.2
8,000	1922.6	389.0	385.4	191.9	98.2	103.9	95.3
10,000	3600.0	766.6	758.6	378.6	190.5	195.0	187.8

Table 9.5: Average execution time (sec) of *MatrixMult* case on Microsoft Azure

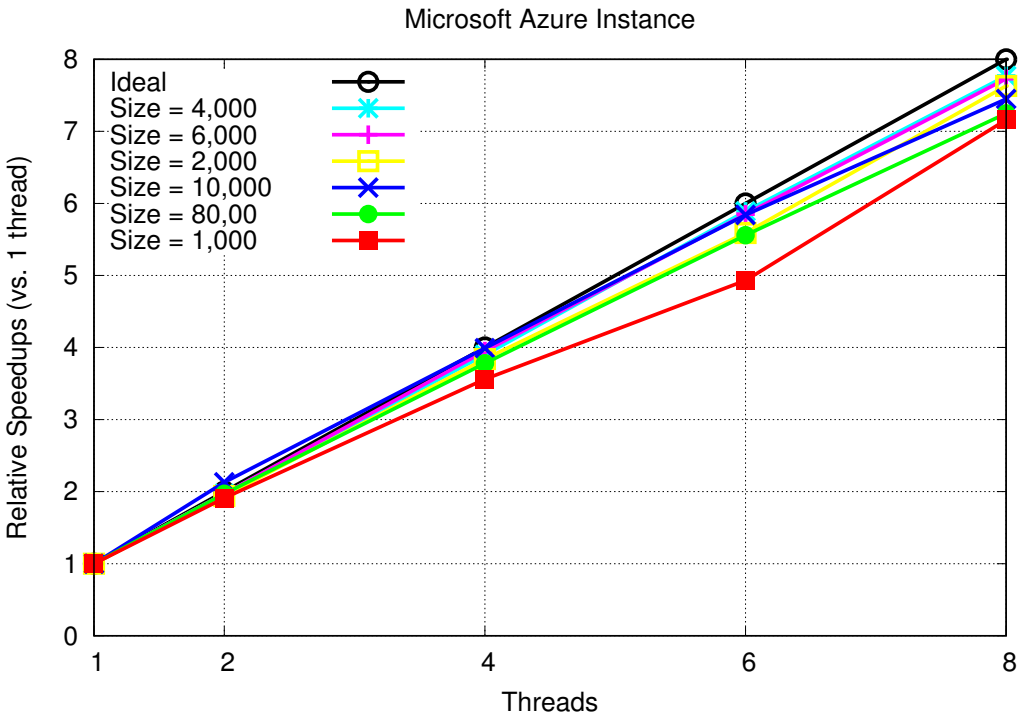
			Polly OpenMP				
Problem Size	GCC	Polly Seq	1 thread	2 thread	4 thread	6 thread	8 thread
1,000	1.4	0.9	0.9	0.5	0.2	0.2	0.1
2,000	24.7	7.3	7.2	3.7	1.9	1.3	0.9
4,000	235.4	57.7	57.2	29.1	14.7	9.7	7.4
6,000	786.2	195.9	193.6	98.3	49.0	33.1	25.1
8,000	1868.1	461.6	456.2	231.0	120.7	82.1	62.9
10,000	3600.0	847.7	896.0	421.3	224.4	153.5	120.3



(a) Standalone Machine (Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz, 4 cores)



(b) AWS EC2 c4.2xlarge (Intel(R) Xeon(R) CPU E5-2666 v3 @ 2.90GHz, 4 cores)



(c) MS Azure F8s (Intel(R) Xeon(R) CPU E5-2673 v3@ 2.40GHz, 8cores)

Figure 9.5: Relative speedups of Polly OpenMP *MatrixMult* program

In conclusion, the benchmark results show that:

- The matrix multiplication has time complexity $O(n^3)$, and thus the execution time greatly increases with matrix sizes.
- Polly sequential code runs at least 4.0x faster than GCC and 1% slower than single threaded Polly OpenMP code, because it optimises data locality of loop iterations and improves the overall performance.
- Polly OpenMP code speeds up the parallel execution and achieves a speed-up at a factor of almost the number of threads until the number of cores is reached.

9.3 Cilk Plus Task Parallelism

`Cilk Plus` (Halpern, 2012) is an extension to C/C++ and makes use of fork-join parallelism in a sequential program. It uses `cilk_spawn` to indicate the

tasks which can be safely run in parallel and `cilk_sync` to set up a barrier to stop the current execution until all the spawned tasks has been completed. We will illustrate Cilk Plus parallelism with *MergeSort* example.

```
1  // Slice an array and perform merge sort on it
2  int* mergesort(int* items, int items_size, int start, int end){
3    if(start +1 < end){
4    int pivot = (start + end)/2;
5    // Slice 'items' into lhs array
6    lhs = slice(items, start, pivot);
7    if(items_size>=1000){
8      // Perform merge sort on lhs array with spawn threads
9      lhs = cilk_spawn mergesort(items, pivot - start, 0, pivot);
10   }else{
11     // Run merge sort in sequential
12     lhs = mergesort(items, pivot - start, 0, pivot);
13   }
14   // Slice 'items' into rhs array
15   rhs = slice(items, pivot, end);
16   if(items_size>=1000){
17     // Perform merge sort on rhs array
18     rhs = cilk_spawn mergesort(items, end - pivot, 0, end);
19   }else{
20     rhs = mergesort(items, end - pivot, 0, end);
21   }
22
23   cilk_sync;
24   // Merge the lhs and rhs arrays
25   while(i< (end -start) && l < (pivot - start)
26         && r < (end - pivot)){
27     ....
28   }
29   return items;// Return the sorted array
30 }
```

Listing 9.3: A hybrid Cilk Plus and sequential merge sort program

Example 9.2 *Function mergesort (see Listing 9.3) combines sequential and Cilk Plus execution. The program sorts the small-size array in sequential and then creates threads to run the sorting on large-sized array in parallel, so that the overheads of Cilk Plus run-time can be reduced.*

The Cilk Plus version of *mergesort* function recursively spawns one thread for each call to perform merge sort on the input array and return the ordered output array. Each spawned function call (Sukha, 2015) is handled with a `stack frame`. The stack frame contains variables, subroutine and passing parameters, and is pushed into the double-ended queue (`deque`) to wait for worker threads to execute.

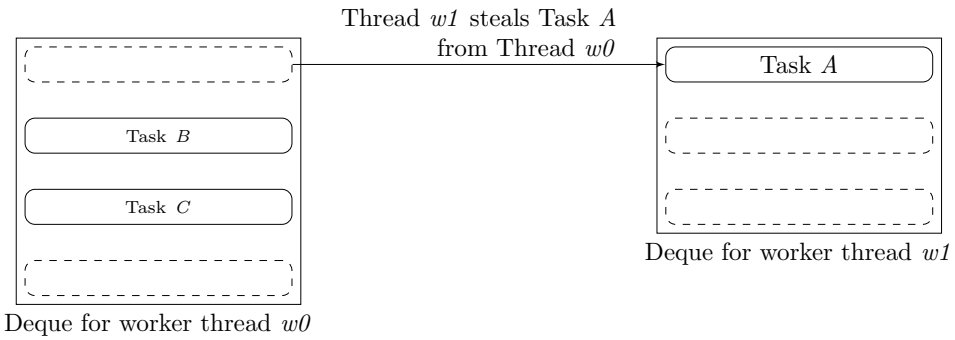


Figure 9.6: Cilk Plus work-stealing task parallelism

Each worker thread has its own deque but allows to take/steal one of the stack frames from the deque of other worker thread. Consider the example in Figure 9.6. The Cilk Plus run-time creates one stack frame for each function call (i.e. A, B and C stack frames). When the deque of work thread *w1* becomes empty, thread *w1* takes frame A from the head of deque *w0* and starts processing the task. By stealing work from a busy thread, Cilk plus run-time keeps all threads busy and runs tasks asynchronously to reduce waiting time in a multi-threaded environment and improve the performance.

9.3.1 Performance Evaluation

The sequential merge sort C program is rewritten as parallel Cilk code to spawn and run the recursive sorting function in parallel and set up a synchronised barrier prior to the merging phase. We experiment three kinds of implementations:

- `Seq` code runs merge sorting in sequential.
- `Cilk Plus` code runs merge sorting in parallel.
- `Cilk Plus + Seq` code spawns a thread to run merge sort function concurrently when array size is larger than 1000. Otherwise, it runs merge sort function in sequential.

Table 9.6: Average execution time (seconds) of Cilk Plus *mergesort* program on standalone machine

		Cilk Plus			
Problem Size	**Seq**	**1 thread**	**2 threads**	**4 threads**	**8 threads**
100,000,000	26.07	32.96	19.90	11.84	10.36
200,000,000	53.29	67.12	40.30	23.93	21.09
300,000,000	82.28	101.62	61.45	36.42	32.32
		Cilk Plus + Seq			
Problem Size	**Seq**	**1 thread**	**2 threads**	**4 threads**	**8 threads**
100,000,000	26.07	28.1	17.4	10.6	9.3
200,000,000	53.29	57.4	35.4	21.5	19.0
300,000,000	82.28	87.5	53.9	32.4	29.0

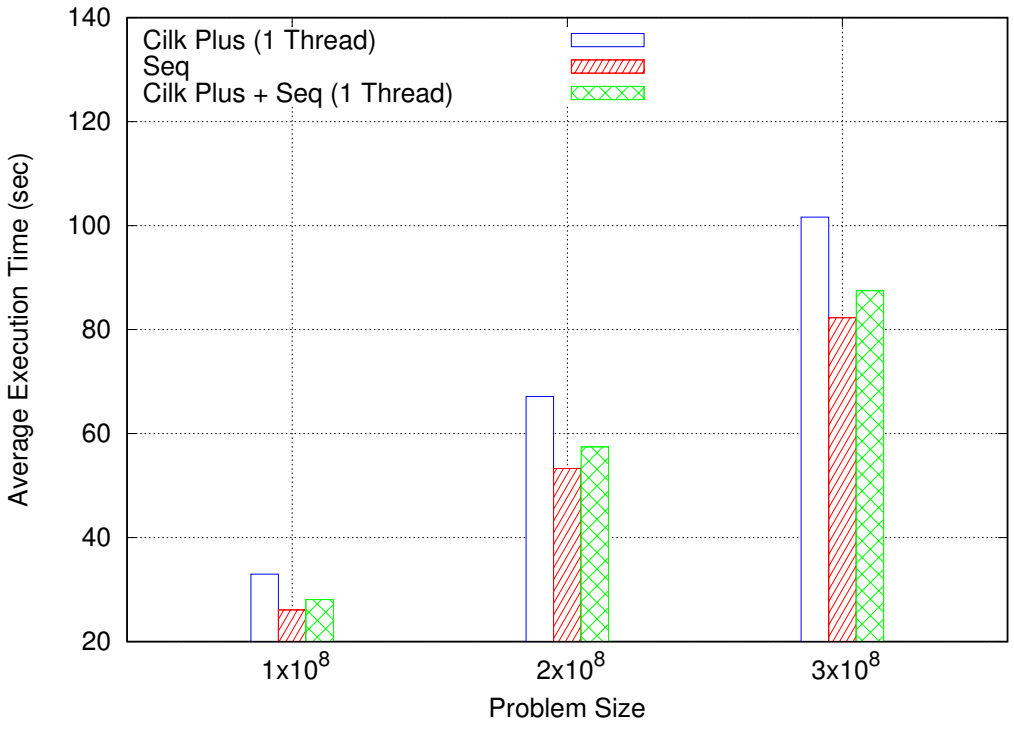


Figure 9.7: Average execution time of Cilk Plus *mergesort* program on standalone machine

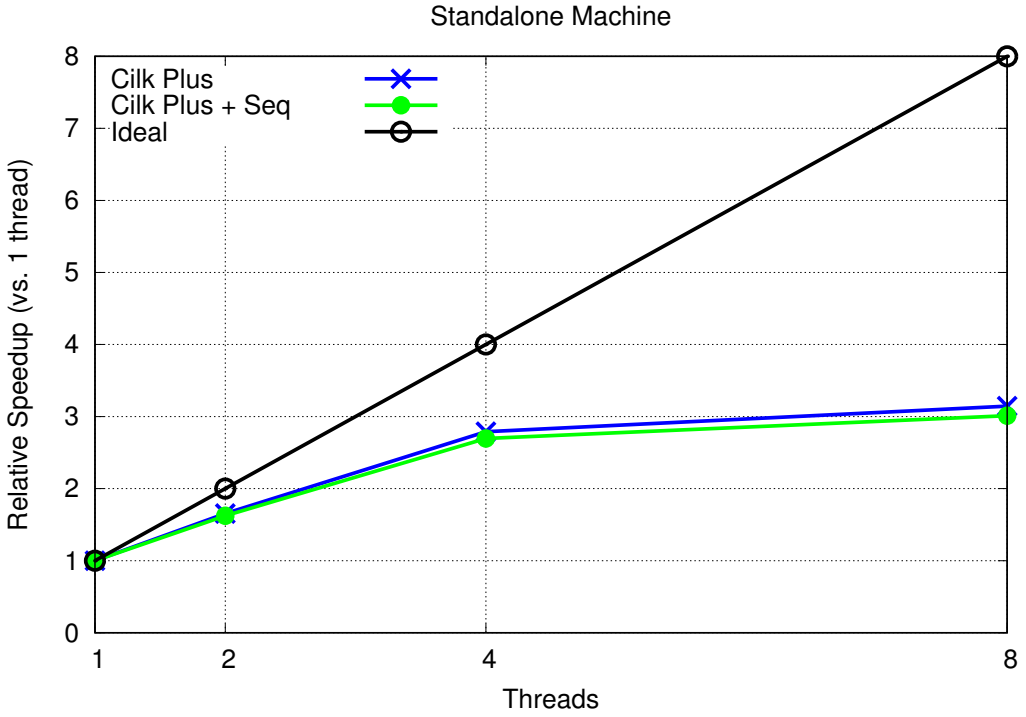


Figure 9.8: Relative speed-ups of Cilk Plus *mergesort* program on standalone machine (problem size: 300,000,000)

Performance on standalone machine The results in Table 9.6 show, with a single thread the sequential (*Seq*) code has the fastest execution time, followed by the combined Cilk Plus and sequential (*CilkPlus* + *Seq*) code. Cilk Plus-only code has the slowest execution. Relative speed-ups in Figure 9.8 shows that the pure and combined Cilk Plus code both improve the speedups with increase of thread numbers and achieves 3.18 speedup with 8 threads over the single threaded implementation.

Performance on virtual machine We benchmark Cilk Plus *mergesort* program on virtual machines of Amazon Elastic Compute Cloud (EC2) and Google Cloud Platform to assess the parallel computing power of Cilk Plus run-time. The specification of these virtual machines are:

- Standalone machine: Intel i7-4770 CPU (@ 3.40GHz, 4 cores) and 16GB
- `Amazon EC2` instance: Intel(R) Xeon(R) CPU E5-2666 v3 (@ 2.90GHz,

8 cores) and 30 GB

- `Google Cloud` instance: Intel(R) Xeon(R) CPU (@ 2.20GHz, 8 cores) and 16 GB

Table 9.7: Average execution time (seconds) of Cilk Plus *mergesort* program on 8-core (up to 16-threads) AWS EC2 machine (Intel(R) Xeon(R) CPU E5-2666 v3 @ 2.90GHz, 30 GB memory)

		Cilk Plus					
Problem Size	**Seq**	**1 thread**	**2 threads**	**4 threads**	**8 threads**	**12 threads**	**16 threads**
100,000,000	32.51	41.02	25.19	14.42	8.99	8.65	8.61
200,000,000	66.76	83.66	51.89	29.27	18.04	17.51	17.46
300,000,000	103.41	126.96	78.99	44.62	27.20	26.42	26.00
		Cilk Plus + Seq					
Problem Size	**Seq**	**1 thread**	**2 threads**	**4 threads**	**8 threads**	**12 threads**	**16 threads**
100,000,000	32.51	34.99	22.05	13.03	8.44	8.07	8.07
200,000,000	66.76	71.77	45.42	26.54	16.84	16.34	16.37
300,000,000	103.41	109.56	68.44	39.87	25.16	24.61	24.34

Table 9.8: Average execution time (seconds) of *mergesort* Cilk Plus program on 8-core (up to 16 threads) Google Cloud machine (Intel(R) Xeon(R) CPU @ 2.20GHz and 16 GB memory)

Problem Size	**Seq**	**Cilk Plus**					
		1 thread	**2 threads**	**4 threads**	**8 threads**	**12 threads**	**16 threads**
100,000,000	37.99	48.39	32.82	20.63	13.51	10.04	9.90
200,000,000	78.23	97.94	66.80	44.04	27.91	20.29	19.83
300,000,000	121.13	148.20	97.50	68.51	41.63	30.22	30.06
		Cilk Plus + Seq					
		1 thread	**2 threads**	**4 threads**	**8 threads**	**12 threads**	**16 threads**
100,000,000		40.8	30.0	20.0	13.5	9.26	9.36
200,000,000		84.0	55.7	36.4	25.9	18.59	19.06
300,000,000		127.9	96.5	55.9	40.8	28.22	28.08

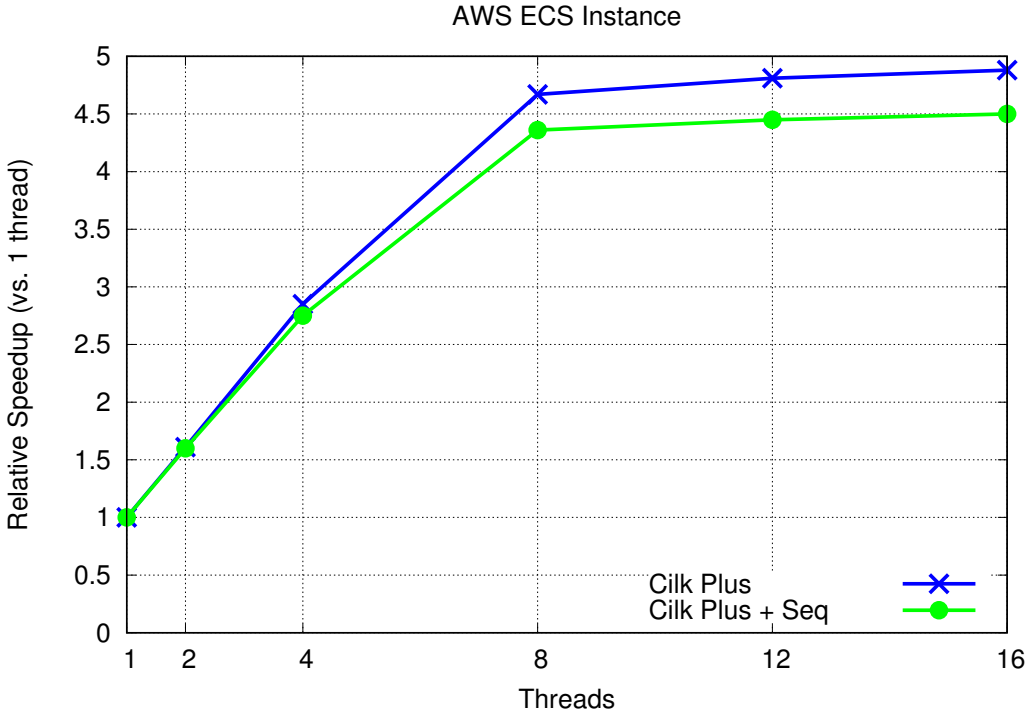


Figure 9.9: Relative speed-up of *mergesort* Cilk Plus program on 8-core (up to 16 threads) AWS EC2 machine (Problem Size: 300 million)

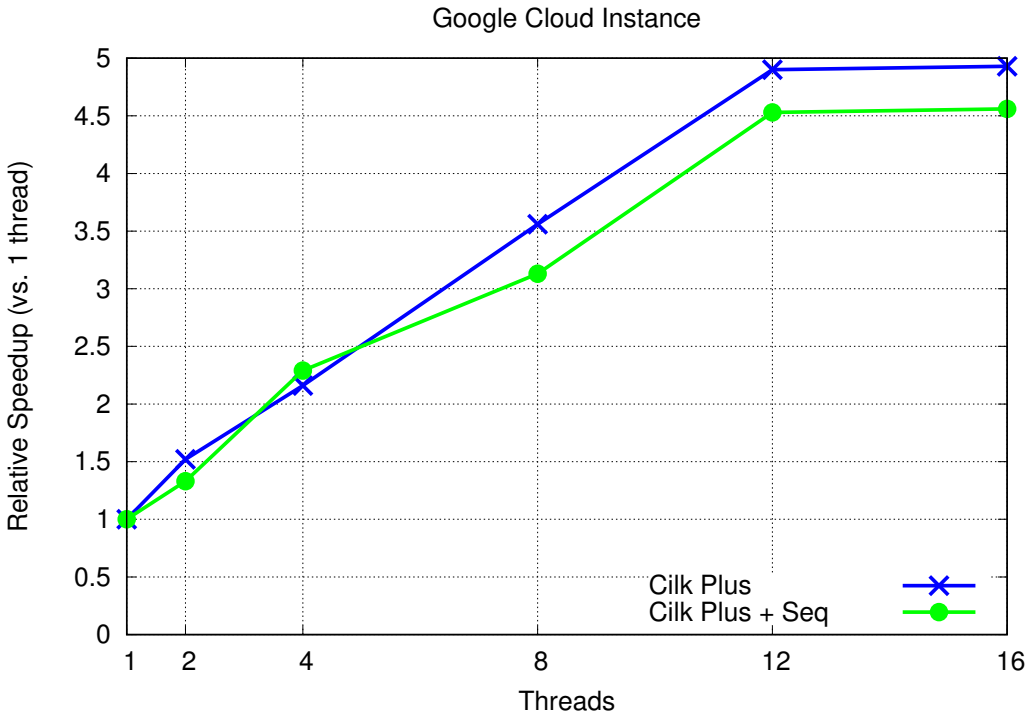


Figure 9.10: Relative speed-up of *mergesort* Cilk Plus program on 8-core (up to 16 threads) Google Cloud machine (problem size: 300 million)

By using 2 or more threads, the parallel Cilk Plus code outperforms the sequential code, and the hybrid *CilkPlus + Seq* code has a better execution time than pure Cilk Plus. However, the pure Cilk Plus code has a slightly better performance scalability over hybrid *CilkPlus + Seq* in both AWS and Google virtual machines. Both of the code can scale the speed up to 8 and 12 threads on AWS EC2 and Google Cloud machines respectively.

9.4 Case Study: Coin Game

We use coin game test case (see Section 8.3) to benchmark the parallelism and performance of Polly compiler, OpenMP and Cilk Plus code. The parallel part of coin game program is as follows. We can divide the coin game into N steps and solve each step sequentially and then keeps track of all results. By re-using the moves from previous step, we can reduce expensive overheads of re-computation and speed up the execution.

```
1  // Use dynamic programming to find moves for Alice
2  function findMoves(int[] moves, int n) -> int[]:
3    int s = 0
4    while s < n: // 0<= s < n
5      int i = 0
6      while i < n -s: // 0 <= i < n -s
7        int j = i + s // j = i + s
8        int y = moves[(i + 1)*n + (j-1)] // y = moves[i+1][j-1]
9        int x = moves[(i + 2)*n + j] // x = moves[i+2][j]
10       int z = moves[i*n + (j-2)] // z = moves[i][j-2]
11       moves[i*n+j] = max(i + min(x, y), j + min(y, z))
12       i = i + 1// End of i,j loop
13     s = s + 1 // End of s loop
14   return moves
15 method main(System.Console sys):
16   int n = Int.parse(sys.args[0])
17   int[] moves = [0;(n+2)*(n+2)]// Increase the move array size to avoid
       wrapping
18   moves = findMoves(moves, n) // Find the moves for Alice
19   int sum_alice = moves[n-1] // Final result of Alice
```

Listing 9.4: Coin game Whiley program

Listing 9.4 shows the Whiley code and the inner loop of $findMoves$ function does not include any if-else branch as we extend the array size and find the maximal and minimal values by using specific macro code (Anderson, 2005): `max(a, b) = a ^ ((a ^ b) & - (a < b))` and `min(a, b) = b ^ ((a ^ b) &`

```
-(a < b)).
```

Table 9.9: Results of *MOVES* arrays in coin game program

$s = 0$	$MOVES(0,0) = 0$ $MOVES(1,1) = 1$ $MOVES(2,2) = 2$
	$MOVES(3,3) = 3$ $MOVES(4,4) = 4$
$s = 1$	$MOVES(0,1) = 1$ $MOVES(1,2) = 2$ $MOVES(2,3) = 3$
	$MOVES(3,4) = 4$
$s = 2$	$MOVES(0,2) = 2$ $MOVES(1,3) = 4$ $MOVES(2,4) = 6$
$s = 3$	$MOVES(0,3) = 4$ $MOVES(1,4) = 6$
$s = 4$	$MOVES(0,4) = 6$

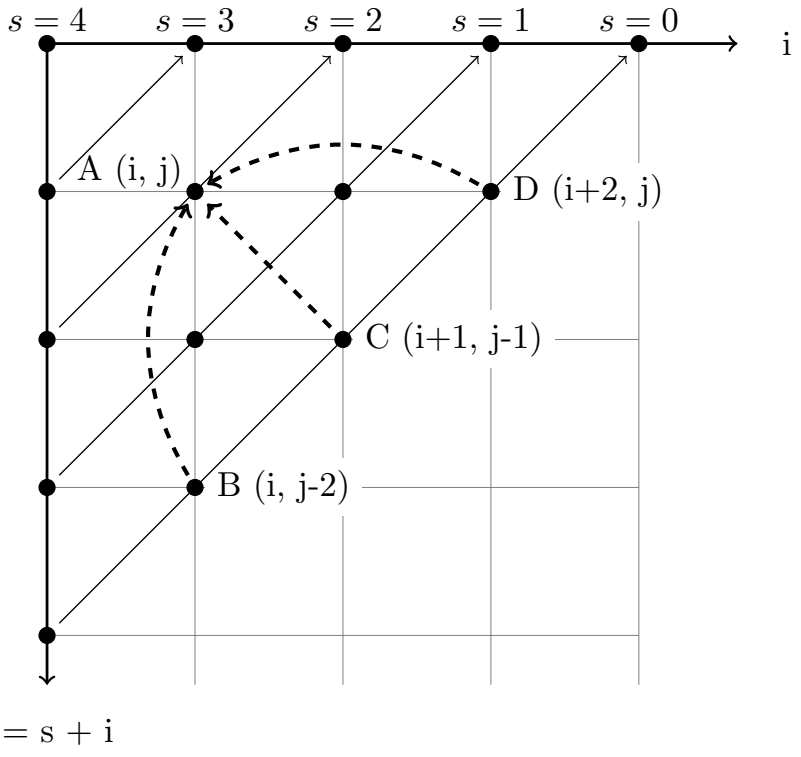


Figure 9.11: Iteration space of the loop in coin game program

Let us consider the coin game with 5 coins $coins := \{0, 1, 2, 3, 4\}$ and all the moves for step $s \in \{0 \ldots 4\}$ are listed in Table 9.9. We draw out the iteration spaces (see Figure 9.11) on the grid chart. Each dot is the move and each diagonal line represents the move for step s.

Each move depends on three neighbouring moves, e.g. the dynamic programming calculates the best move for $A(1,3)$ by reading the move from $B(1,1)$, $C(2,2)$ and $D(3,3)$ on the diagonal line of $s = 0$, and then obtain the maximal scores for Alice's move by applying the below equation to :

$$MOVES[i][j] = max(C_i + min(MOVES[i+2][j],\ MOVES[i+1][j-1]),$$
$$C_j + min(MOVES[i+1][j-1],\ MOVES[i][j-2])) \tag{9.1}$$

9.4.1 OpenMP Parallel For

From Figure 9.11, we notice on the same diagonal line each i iteration exhibits no dependency with other variables. Also, the inner loop does not have to preserve the order because its calculation relies only on the moves of previous iterations, which have been computed and stored in the array.

```
#include "omp.h"
// Find the moves in parallel
int* findMoves(int* moves, int n){
  for(int s = 0; s<n; s++){
    // Use parallel worksharing OpenMP construct
    #pragma omp parallel for
    for(int i = 0;i<n-s; i++){
      int j = i+s; // 'j' variable depends on 's'
      int y = moves[(i+1)*n + j-1];// moves[i+1][j-1]
      int x = moves[(i+2)*n + j];// moves[i+2][j]
      int z = moves[i*n + (j-2)];// moves[i][j-2]
      moves[(i*n)+j]=max(i+(min(x, y)), j+ (min(y, z)));
    }
  }
  return moves;
}
```

Listing 9.5: OpenMP parallel for loop in coin game code

That exposes a potential parallelism for the inner loop and splits i iterations into a team of threads so that each thread can handle one part of the loop independently and in parallel. So we use `omp parallel for` OpenMP worksharing construct to share the iterations of i loop across different threads and to execute in parallel, as shown in Listing 9.5. Note that each move in iteration s depends on the previous iterations (see Figure 9.11). As a result of explicit data dependency, we can not parallelise the outer loop iteration s.

9.4.2 Cilk Plus For

```
#include <cilk/cilk.h>
int* findMoves(int* moves, int n){
  for(int s = 0; s<n; s++){
    // Use default grain size min(2048, ceil(n−s / (8 ∗ threads)))
    cilk_for(int i = 0;i<n-s; i++){
      int j = i+s;
      int y = moves[(i+1)*n+j-1];
      int x = moves[(i+2)*n+j];
      int z = moves[(i*n)+j-2];
      moves[(i*n)+j]=max(i+(min(x, y)), j+ (min(y, z)));
    }
  }
  return moves;
}
```

Listing 9.6: Cilk PLus parallel for loop in coin game

We use `cilk_for` keyword to parallelise the inner loop and run the moves of same s in a team of threads. As Cilk Plus uses divide-and-conquer technique, Intel Cilk Plus run-time divides i iterations into two halves, where each part roughly has equal length, and then recursively sub-divides each part into half until each sub-part is less than grain size. In this example, we use default equation for choosing grain size:

$$\text{cilk grainsize} = min(2048, \frac{N}{8 \times p})$$

where N is loop iterations and p is the number of threads.

Then the work-stealing scheduler automatically distributes the work to available cores, keeps all worker threads busy and reaches workload-balance. Each worker thread has one queue to store all its unfinished work. When a new task comes in, the worker pushes this task to the head of its de-queue. And then the worker takes out one work from the bottom of de-queue and start executing it.

Once the de-queue becomes empty, the worker randomly chooses another worker and steals one of its work from the head of its de-queue so that no worker is idle for most of time. Randomised work-stealing algorithm (Blumofe and Leiserson, 1999) has been mathematically proven to be more efficient in join-fork computation.

9.4.3 Benchmark Results

Our benchmark program creates an array of coins with given size, and each coin value is the same value as array index $coins[i] = i$. In doing so, we can ensure all our benchmarks produce the same output.

9.4.3.1 Performance Evaluation on Standalone Machine

We benchmark the code on Ubuntu 16.04 standalone machine with Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz and 16 GB. C compilers are used to compile C program into parallel executable, including GCC 5.4 and Polly compiler.

Table 9.10: Average execution Time (seconds) of parallel *coin game* programs on 4-core (up-to 8 threads) standalone machine (Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz and 16 GB memory)

		Polly OpenMP			
Problem Size	**Seq**	**1 thread**	**2 threads**	**4 threads**	**8 threads**
10,000	0.2892	0.310	0.298	0.300	0.298
20,000	1.17	1.19	1.20	1.18	1.20
30,000	4.37	4.37	4.38	4.41	2.75
40,000	5.03	4.98	4.94	4.92	4.93

Polly Compiler We use Polly to automatically optimise the sequential code of coin game, produced by our code generator with copy and deallocation analysis enabled, and then exploit the parallelism and generate parallel OpenMP code. Table 9.10 shows the benchmark results on 4 cores (8 hyper-thread) machine, and that Polly parallel code has no speed-ups over Polly sequential code, and does not scale up with thread numbers.

OpenMP and Cilk Plus Parallelism We benchmark parallel coin game in OpenMP and Cilk Plus code, and each code is compiled into executable

with GCC compiler (v.5.4.0) commands:

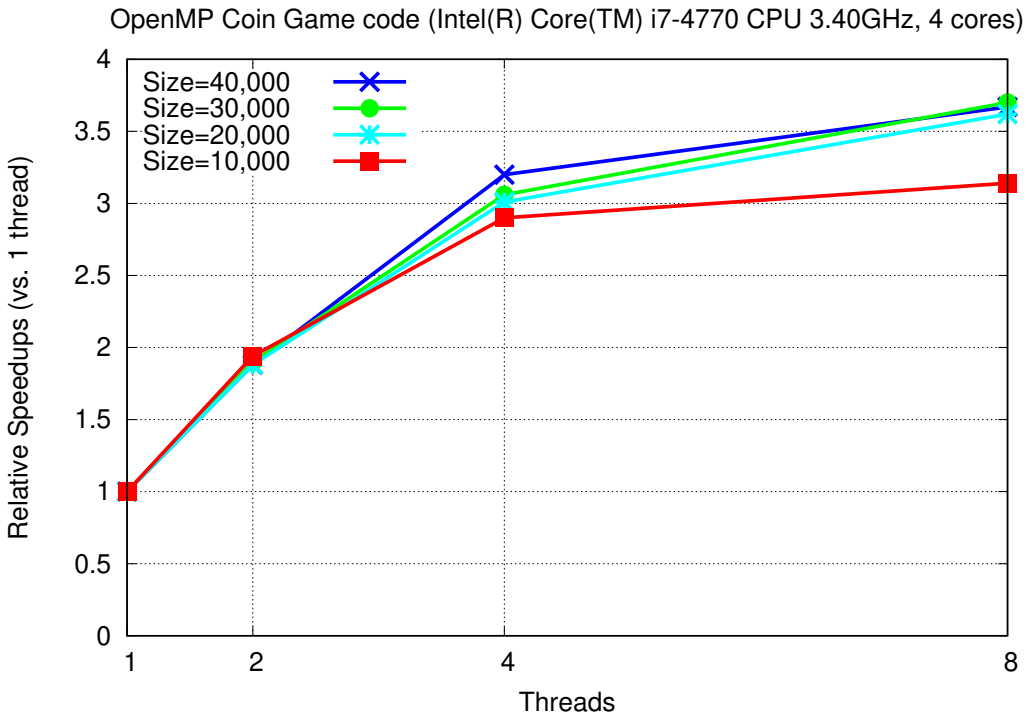
- Sequential code: `gcc -O0`
- OpenMP code: `gcc -fopenmp -O0`
- Cilk Plus code: `gcc -fcilkplus -O0 -lcilkrts`

Then we specify the number of threads to use in parallel region using `OMP_NUM_THREADS` and `CILK_NWORKERS` environment variables. Then each benchmark is repeatedly run for 10 times on 4-core (up to 8 threads) machine and the execution times are averaged.

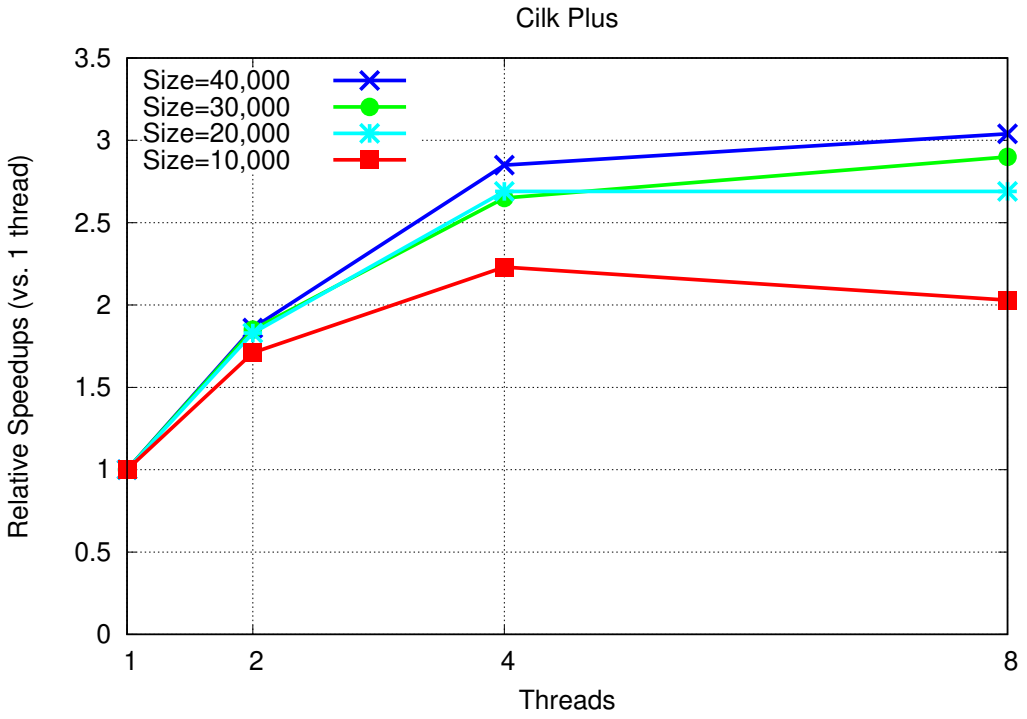
Table 9.11: Average execution time (seconds) of parallel *coin game* programs on 4-core standalone machine (Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz and 16 GB memory)

		OpenMP			
Problem Size	**Seq (-O0)**	**1 thread**	**2 thread**	**4 thread**	**8 thread**
10,000	0.876	0.916	0.473	0.316	0.291
20,000	3.579	3.71	1.98	1.23	1.03
30,000	7.960	8.36	4.40	2.73	2.26
40,000	14.264	14.79	7.87	4.62	4.03
		Cilk Plus			
Problem Size	**Seq (-O0)**	**1 thread**	**2 thread**	**4 thread**	**8 thread**
10,000	0.876	0.902	0.527	0.404	0.444
20,000	3.579	3.72	2.03	1.39	1.39
30,000	7.960	8.20	4.44	3.10	2.83
40,000	14.264	14.65	7.89	5.14	4.81

Table 9.11 shows that both parallel OpenMP and Cilk Plus using a single thread runs roughly 3% $\sim$ 4% slower than sequential code. So the over-head costs of parallelism slightly reduces the program execution.



(a) OpenMP coin game program



(b) Cilk Plus coin game program

Figure 9.12: Relative speedup of parallel coin game programs on 4-core machine (Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz, 4 cores)

Figure 9.12 shows the relative speedups of OpenMP and Cilk Plus code respectively. Both of OpenMP and Cilk Plus code exhibit good performance scalability on large problem ($>= 20,000$). The parallel speedup increases up-to the number of cores, and becomes normal 3.6x and 3.0x speedup on using 8 threads.

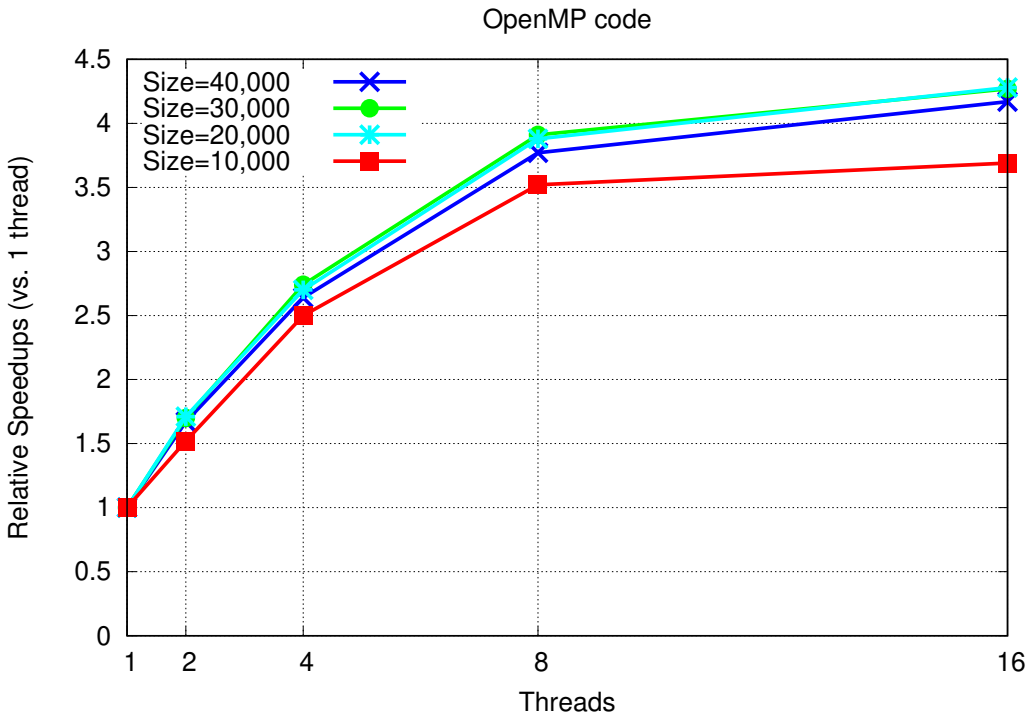
To sum up, Polly compiler has efficient sequential code because its data locality gains speed-ups from cache behaviour, but does not improve the performance with concurrency. OpenMP/Cilk Plus sequential version runs slower than Polly, but can achieve $3.0 \sim 3.6$ speed-ups with concurrency and roughly the same speed at 4/8 threads.

9.4.3.2 Performance Evaluation on Virtual Machine

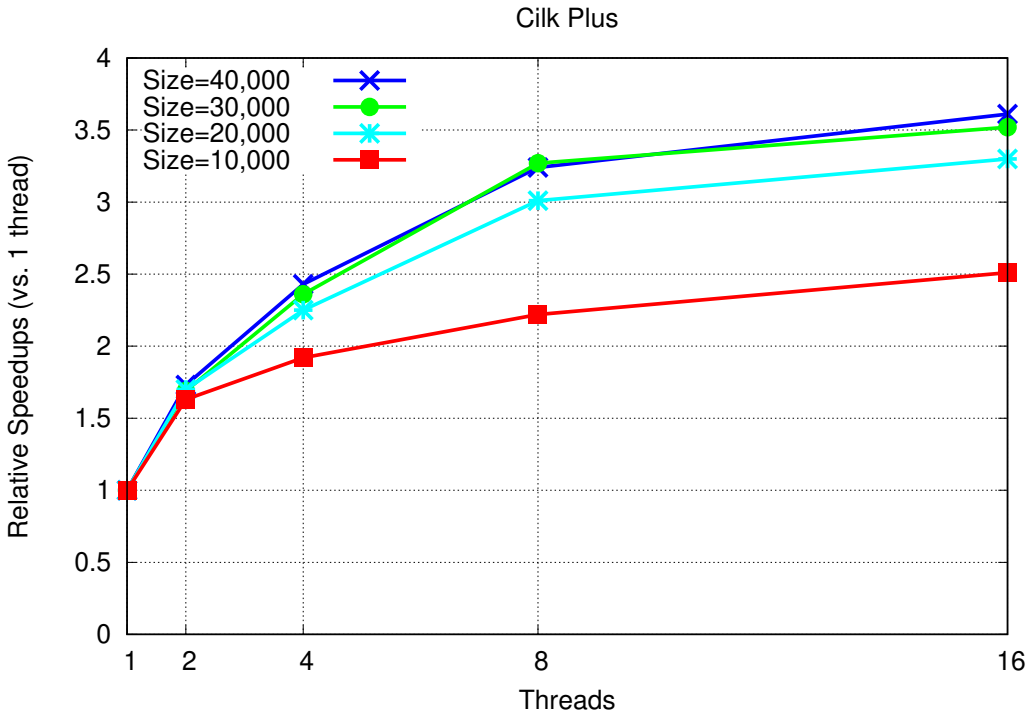
This section shows the benchmark results running coin game on HPC clouds.

Table 9.12: Average execution time (seconds) of parallel coin game code on 8-core (up to 16 threads) Google Virtual Machine (Intel(R) Xeon(R) CPU @ 2.20GHz and 16 GB memory)

			OpenMP			
Problem Size	**Seq**	**1 thread**	**2 thread**	**4 thread**	**8 thread**	**16 thread**
10,000	2.07	2.09	1.37	0.836	0.593	0.565
20,000	10.32	10.92	6.39	4.04	2.81	2.55
30,000	24.37	25.12	14.82	9.17	6.43	5.88
40,000	45.22	41.29	24.76	15.63	10.96	9.91
			Cilk Plus			
Problem Size		**1 thread**	**2 thread**	**4 thread**	**8 thread**	**16 thread**
10,000		2.18	1.34	1.14	0.981	0.868
20,000		10.84	6.39	4.82	3.60	3.28
30,000		24.96	14.73	10.60	7.64	7.08
40,000		40.98	23.66	16.88	12.63	11.34



(a) OpenMP coin game code



(b) Cilk Plus coin game code

Figure 9.13: Relative speed-up of coin game on 8 cores (16 hyper-threads) Google Cloud Machine (Intel(R) Xeon(R) CPU@2.20GHz and 16 GB)

The parallel benchmarks of coin game OpenMP and Cilk Plus reducer programs were performed on 8 core (16 hyper-threads) Google Compute Engine (Ubuntu 16.04, Intel(R) Xeon(R) CPU @ 2.20GHz and 16 GB memory). The benchmarks include GCC 5.4, which has been integrated with Intel Cilk Plus. The compilation options is as follows:

- Sequential code: `gcc -O0`
- OpenMP code: `gcc -fopenmp -O0`
- Cilk Plus code: `gcc -fcilkplus -O0 -lcilkrts`

The detailed benchmark results are listed in Table 9.12.

Figure 9.13 shows benchmark results of OpenMP and Cilk Plus coin game on 8-core Google virtual machine. Both OpenMP and Cilk Plus with a single thread exhibit similarly and even slightly better performance (max = 1.1x speedup) than sequential code at zero optimisation level. With 1 or 2 threads, there is no significant difference between Cilk Plus and OpenMP. Both OpenMP and Cilk Plus gain more than 3.0x speedups with 8 threads and slight better with 16. And OpenMP code has a slightly better performance than Cilk Plus on multi-threaded execution.

OpenMP and Cilk Plus code both improve performance with thread numbers and slightly better with 16. OpenMP code maintains consistent and better scalability over all problem sizes, whereas Cilk Plus has relatively poor scalability on smallest problem.

Conclusion Polly compiler does not parallelise our coin game C program because the loop has implicit data dependency on the shared array, so Polly produces efficient sequential code only. Both of OpenMP and Cilk plus code can be executed in parallel at speed close to linear with the number of threads on multi-threaded standalone and virtual machines. Further, OpenMP has a better parallel efficiency and faster execution than Cilk Plus in coin game case.

9.5 Case Study: LZ77 Compression

We use LZ77 compression (see Section 8.4.1) as test case and experiment the parallel efficiency of Polly compiler, OpenMP and Cilk Plus. We use the below pre-allocated array Whiley program and combine copy elimination and deallocation analysis to produce the sequential C code.

9.5.1 Polly Parallelism

We use Polly compiler to compile the sequential C code into OpenMP parallel code and bench-marked on standalone machine to measure the speed-ups from Polly optimisation and parallelism.

Table 9.13: Average execution time (seconds) of Polly LZ77 compression program on 4-core standalone machine (Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz, 16 GB memory)

Problem Size	**Polly Seq**	**Polly OpenMP**			
		1 thread	2 thread	4 thread	8 thread
large1x (0.57 MB)	0.084	0.092	0.085	0.086	0.082
large2x (1.1 MB)	0.154	0.195	0.152	0.152	0.151
large4x (2.3 MB)	0.296	0.303	0.285	0.300	0.301
large8x (4.6 MB)	0.546	0.578	0.559	0.554	0.542
large16x (9.2 MB)	1.07	1.07	1.08	1.09	1.08
large32x (18.4 MB)	2.14	2.19	2.13	2.12	2.12
large64x (36.8 MB)	4.19	4.24	4.19	4.22	4.20
large128x (73.6 MB)	8.42	8.43	8.39	8.45	8.40
large256x (147.2 MB)	16.82	16.74	16.83	16.83	16.79

Table 9.13 shows there is no significant speedups on Polly optimisation and parallelism. Polly compiler reports the following messages: an affine expression can not be derived from the loop bound in function *match*.

```
1  // Find the matched entry with affine loop bound
2  function match(byte[] data, nat offset, nat end) -> (int length)
3      ensures 0 <= length && length <= 255:
4      nat pos = end
5      nat len = 0
6      while offset < pos && pos < |data| && len < 255
7          && data[offset] == data[pos]:
8          offset = offset + 1
9          pos = pos + 1
10         len = len + 1
11     return len
```

Listing 9.7: Function *match* in LZ77 compression Whiley program

The loop bound consists of four conditions. The first three can individually form a valid affine expression using reference variables *offset*, *pos* or *len*. But the last condition $data[offset] == data[pos]$ checks the value of array *data* and would terminate the loop earlier than expected. Because of unpredictable behaviours, the loop bound can not be expressed with a linear relation with a reference variable. Thus, Polly compiler can not optimise the loop and run it in parallel.

9.5.2 OpenMP Map/Reduce Code

OpenMP (Arif and Vandierendonck, 2015) provides multiple constructs to facilitate map-reduce programming. In map phase, `parallel for` clause can be used to partition a large loop iterations and then spawn a team of threads to run each part concurrently. In reduce phase, OpenMP allow user-defined reduction operation to combine all the intermediate values into a single result.

```
1  // Find the longest match for current position 'pos'
2  Match* findLongestMatch(BYTE* data, int pos){
3    int bestLen=0, bestOffset=0, offset;
4    int start=max(pos-255, 0);
5    for(offset =start;offset<pos;offset++){// The loop can be parallelised
6      // Call function match to find the match for each 'offset'
7      int len = match(data, offset, pos);
8      if (len > bestLen){
9        bestLen = len;
10       bestOffset = pos - offset;
11     }
12   }
13   Match* match = malloc(sizeof(Match));
14   match -> len = bestLen;
15   match -> offset = bestOffset;
16   return match;
17 }
```

Listing 9.8: Sequential code of searching the match in LZ77 Compression

We illustrate OpenMP map-reduce style programming with the procedure that finds the longest match in LZ77 compression. The search goes through array *data* and finds the longest match from the string occurring earlier and then outputs one length-offset match (two bytes). The sequential program is as follows.

Position	Length-Offset Pair	Best match
POS:0	bestLen:0 bestOffset:0	(0, 'A')
POS:1	bestLen:1 bestOffset:1	(1, 1)
POS:2	bestLen:0 bestOffset:0	(0, 'C')
POS:3	bestLen:3 bestOffset:4	(3, 4)
POS:7	bestLen:0 bestOffset:0	(0, 'B')
POS:8	bestLen:3 bestOffset:3	(3, 3)
POS:11	bestLen:2 bestOffset:11	(11, 2)
POS:13	bestLen:2 bestOffset:12	(12, 2)

Table 9.14: Best match of string *AACAACABCABAAAC*

The sequential code keeps a window size of data buffers (256 bytes in this case) and then goes through one offset after another to search for the best match, that has the longest length and offset. For example, 'AACAACAB-CABAAAC' can be encoded as above table. The procedure of finding a match can be encoded as local map tasks, and searching for the best match then can be implemented as a single reduce task, illustrated as below.

The OpenMP map/reduce program creates a number of threads to find the longest match length concurrently. The program contains initialise, map and reduce phases.

- *Initialise phase* create two arrays *localLen* and *localOffset* to store the local optimal match in each thread, and use `omp single` clause to ensure these array values are initialised only once by a single thread.

- *Map phase* partitions the offset iterations in to a number of sub-tasks, and solves each sub-task in each thread to find the best local optimal match.

- *Reduce phase* obtains the global optimal match.

```
int64_t bestLen, bestOffset;
int64_t* localLen;
int64_t* localOffset;
int numofthreads;
#pragma omp parallel default(shared)
{
  // Initialize local length and offset
  #pragma omp single
  {
    numofthreads= omp_get_num_threads();
    localLen = malloc(numofthreads*sizeof(int64_t));
    localOffset = malloc(numofthreads*sizeof(int64_t));
    // Initialize local_len and local_offset
    for(int i =0; i <numofthreads;i++){
      localLen[i] = 0;
      localOffset[i] = 0;
    }
  }
  // Map phase
  int tid = omp_get_thread_num(); // Thread ID
  #pragma omp for
  for(offset = start;offset<pos;offset++){
    //Private variable to store the found match length
    int64_t len = match(data, offset, pos);// local variable
    // Find local optimal length and offset
    if(len > localLen[tid]){
      localLen[tid] = len;
      localOffset[tid] = pos - offset;
    }
  }
  // Reduce phase
  #pragma omp single
  {
    // Find the global optimal length and offset
    for(int i =0; i <numofthreads;i++){
      if(localLen[i]>bestLen){
        bestLen = localLen[i];
        bestOffset = localOffset[i];
      }
    }
  }
}
free(localLen);
free(localOffset);
```

Listing 9.9: OpenMP map-reduce LZ77 compression code

Listing 9.9 shows the OpenMP LZ77 compression program. the some changes for OpenMP map/reduce program is as follows.

Map phase shares all the array variables in OpenMP parallel region to avoid race conditions, except for the match length variable *len*. As the offset

iteration space is split into several parts, each thread can take a subset of offsets and compute their best match independently. By privatising the match length to each thread, we can avoid expensive synchronisation overhead, e.g. using `omp ordered` clause to enforce the multi-threads executing in the same order as the sequential one. And we use `omp for` work-sharing clause in map phase to distribute the offset iterations to all the available threads, where each updates the local arrays *localLen* and *localOffset* indexed by its thread number. *Reduce phase* use `omp single` clause to collectively obtain the global match with master thread.

We illustrate the OpenMP map/reduce program to find the best match at position 3 using 3 threads. First, we can use `omp parallel num_threads(3)` clause to specify the number of threads and `omp single` code region initialises the local optimal array values only once.

Table 9.15: Sample outputs of LZ77 OpenMP map/reduce program at position 3 using 3 threads

	Thread ID	Local variables	Shared variables
POS: 3	0	offset: 0 len: 4	localOffset[0]: 3 localLen[0] :4
	1	offset: 1 len: 1	localOffset[1]: 2 localLen[1]: 1
	2	offset: 2 len: 0	localOffset[2]: 0 localLen[2]: 0
	bestOffset: 3 bestLen: 4		

Second, we use `omp for` clause to divide the loop iterations into three parts, so that each thread processes only one part simultaneously. Table 9.15 shows the local and global matches found by all threads. Each thread takes one part of *offset* iterations as input to search for the longest match, so thread 0 finds the match for *offset = 0* and thread 1 searches the match for *offset = 1* and so on. Once it finds a new match of longer length and then stores the match in shared arrays *localOffset* and *localLen* where each thread only allows to access

one array element at the index of its distinct thread id to avoid race condition.

Third, the reducer waits until all mapper tasks are completed, and then starts iterating array *localLen* and obtain the longest match. And by specifying `omp single` clause, we can ensure the reducer executes as the sequential one.

9.5.3 Cilk Plus Reducer

The offset loop in LZ77 program can be executed in parallel by using Cilk Plus `cilk_for` keyword. In doing so, Intel Cilk Plus compiler and run-time uses divide and conquer technique to split all the offset iterations into two halves recursively until each child thread is busy.

```
1  void Match_init(Match* m) {// Initialize a match to be empty
2    m->len=0;
3    m->offset=0;
4  }
5  void identity_Match(void* reducer, void* m)// Reset reducer's value
6  {
7    Match_init((Match*)m);
8  }
9  // Combine two reducer's values into left reducer.
10 void reduce_Match(void* reducer, void* left, void* right)
11 {
12   Match* l_m = (Match*)left;
13   Match* r_m = (Match*)right;
14   if(l_m-> len < r_m-> len){
15     l_m->len = r_m->len;
16     l_m->offset = r_m->offset;
17   }
18 }
19 CILK_C_DECLARE_REDUCER(Match) my_match_reducer =
20         CILK_C_INIT_REDUCER(Match, reduce_Match, identity_Match,
     __cilkrts_hyperobject_noop_destroy);// Define a customised reducer
21 // We register/unregister my_match_reducer with Intel Cilk runtime
22 Match* findLongestMatch(..){
23   ....
24   {
25     Match_init(&REDUCER_VIEW(my_match_reducer));// Initialize the
       reducer
26     // Spawned threads and execute the offset loop
27     cilk_for(int offset = start;offset<pos;offset++){
28       int64_t len = match(data, false, offset, pos);
29       Match* m= &REDUCER_VIEW(my_match_reducer);//Get the reducer
30       if(len > m->len){// Update reducer with a better 'len-offset' pair
31         m->len = _len;
32         m->offset = pos - offset;
33       }
34     }
35     Match* m = &REDUCER_VIEW(my_match_reducer);//Get the reducer
36     bestLen = m->len;
37     bestOffset = m->offset;
38   }
39 }
```

Listing 9.10: LZ77 compression using Cilk Plus reducer

Our Cilk Plus for loop concurrently updates the best match but may cause data race condition. To ensure our match is accessed by a single thread at each time, we introduce `cilk reducer` to serialise the access whilst guaranteeing the execution order. Cilk Plus procedure in Listing 9.10 is described as below:

- Declare a customised reducer in global scope with identity function, reduce and destroy functions
  - `identity_Match` function is used to initialise reducer's value when a thread begins.
  - `reduce_Match` function compares and combines the values of two reducers ($left$ and $right$) into one value ($left$) of a working thread, which has a larger match length.
- Inside *compress* function, register the reducer and enable the run-time to manage the reducer's value during parallel execution.
- Inside *findLongestMatch* function,
  - Retrieve the address of reducer using `REDUCER_VIEW` and initialise its value.
  - Use `cilk_for` to partitions offset iterations and to spawn threads and run the loop concurrently. Inside the loop, we also use `REDUCER_VIEW` to obtain the reducer's value in parallel execution and update it with the best length and offset.
- `__cilkrts_hyperobject_noop_destroy` function enables the run-time to automatically clean up the memory of reducers that are no longer in use.

The use of Cilk Plus reducer (Frigo et al., 2009) ensures the reducers are thread-safe and preserving the execution order with minimal overhead costs in multi-threaded environment.

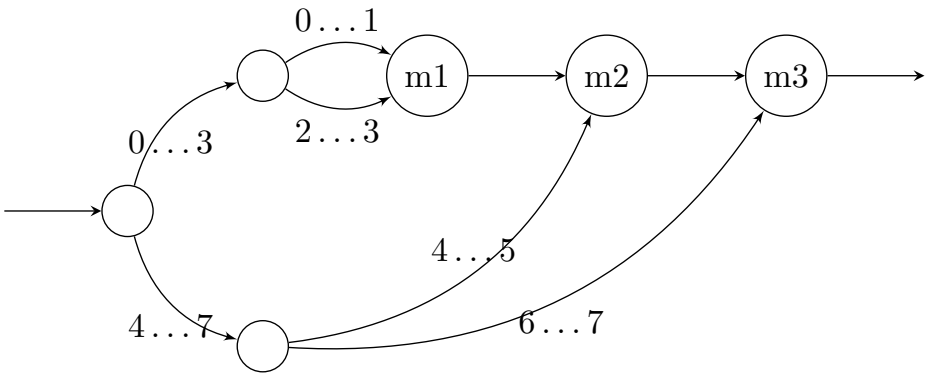


Figure 9.14: Directed Acyclic Graph (DAG) of offset loop iterations ($N = 8$) with 2 threads in LZ77 compression

Table 9.16: Sample outputs of Cilk Plus LZ77 compression at position 8 using 2 threads

	Reducer Id	Local variable	Reducer
POS=3	r0	offset:0 len:0	r0.offset:0 r0.len:0
		offset:1 len:0	r0.offset:0 r0.len:0
	r1	offset:2 len:2	r1.offset:6 r1.len:2
		offset:3 len:0	r1.offset:6 r1.len:2
	r2	offset:4 len:0	r2.offset:0 r2.len:0
		offset:5 len:3	r2.offset:3 r2.len:3
	r3	offset:6 len:0	r3.offset:0 r3.len:0
		offset:7 len:0	r3.offset:0 r3.len:0
	bestOffset:3 bestLen:4		

The data race of shared variable is a common synchronisation error in parallel execution. Traditionally the mutex lock could solve this problem, but usually leads to long delay in data contention and increases extra costs in overhead. In contract to lock-based mechanism, the reducers create a new

instance of lock-free *view* for each spawned thread, so that each strand can manipulate the reducer privately and avoid data sharing and collision. When a strand finishes its task and returns to the parent thread, the reducer applies *reduce* function to merge the reducer's views from two strands and leave one reducer view. The procedure of merging reducers continues until all strands finish executing and leave the final result to the initial reducer's view.

Consider LZ77 compression as an example. The program tries to find the longest match at position 8 with 2 working threads. Cilk Plus run-time divides the offset iterations ($0 \ldots 7$) into two parts ($0 \ldots 3$ and $4 \ldots 7$), and then divides each part into two halves, as shown in Figure 9.14. In DAG graph, each path of a number indicates the offset iterations that each strand needs to process, e.g. $0 \ldots 1$ means the strand finds the best match from iteration 0 to 1.

Table 9.16 shows the value of reducer views in each strand, where *r0* ... *r3* are the reducers for each thread, and *m1* ... *m3* are nodes that each merges the values of two reducers.

- The reducer creates and initialises one private view for each strand so that each spawned thread can store its local optimal match.
- After all strands finish their work, the run-time starts to merge the results in each strand.
  - *m1* merges *r0* and *r1*, and leaves *r1*.
  - *m2* merges *r2* and *r3*, and leaves *r2*.
  - *m3* merges *r1* and *r2*, and leaves *r2*.
- The final best reducer is *r2*(*offset* : *3*, *len* : *3*), so the best offset is three and the longest length is three.

Grain Size `#pragma cilk grainsize` specifies the number of loop iteration that each strand is allow to execute. The default grain size is:

$$\#\text{pragma cilk grainsize} = \text{GRAINSIZE} = min(2048, \frac{N}{8 \times p})$$

where N is loop iterations, and p is the number of threads.

Table 9.17: Grain size varying on large256x (147.2 MB) file

Grain Size	1 Thread	2 Threads	4 Threads	8 Threads
Default	129.4	122.7	149.8	233.3
1	129.5	122.8	149.5	235.9
2	129.6	122.8	149.1	234.7
4	128.7	122.4	149.7	234.8
8	129.4	123.8	150.7	235.5
16	130.2	123.0	148.7	235.3
32	130.2	122.6	148.9	235.3
64	131.4	122.9	149.9	234.4
128	130.3	122.3	149.1	234.5
256	130.5	122.8	149.0	235.0

We vary the number of grain size and set it to be the fixed size, as shown in Table9.17, and measure the speed-up over default size on large256x file. The benchmark results show that, increasing grain size does not give significant speed-ups.

9.5.4 Benchmarks

We use GCC 5.4 to compile OpenMP map/reduce program with `-fopenmp -O0` flag enabled and linked with OpenMP run-time library. And the sequential code which strips off all OpenMP clauses is also compiled at default optimisation level (`-O0`), to make an fair comparison with parallel OpenMP code.

We also use GCC 5.4 to compile Cilk Plus reducer program with `-fcilkplus -O0 -lcilkrts` flags to link the executable with Cilk Plus run-time. Every experiment is repeated for 10 times and the execution time is averaged.

9.5.4.1 Performance Evaluation on Standalone Machine

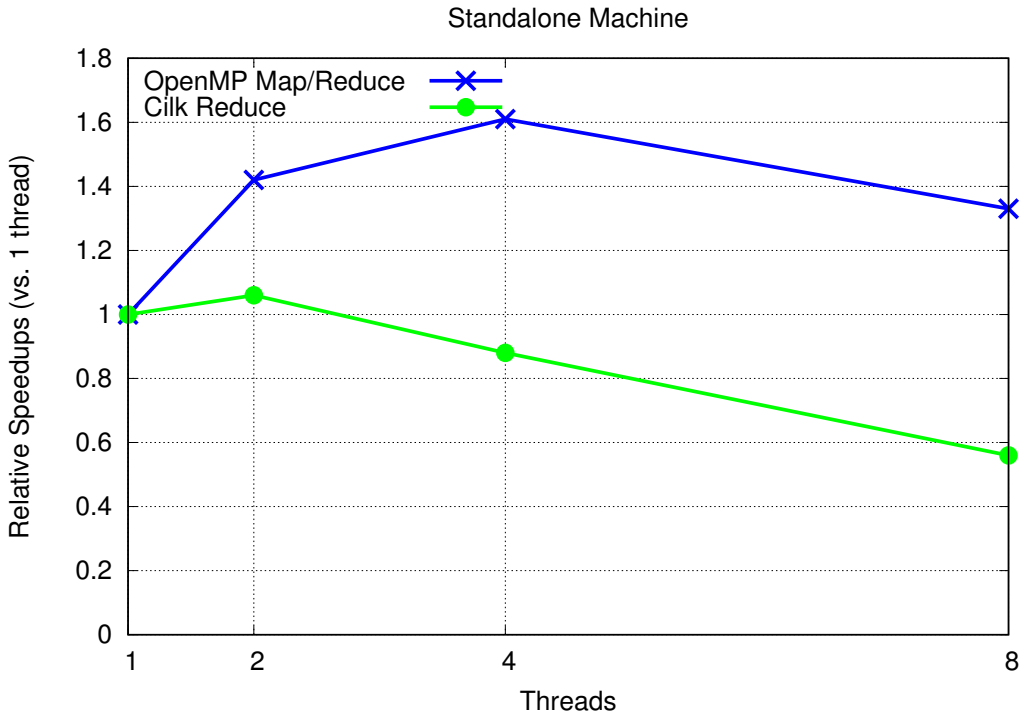


Figure 9.15: Relative Speedup of parallel LZ77 compression program on 4-core (up to 8 threads) standalone machine (Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz and 16 GB memory)

The benchmarks are listed in Appendix Table C.3. Figure 9.15 shows that, on 8-threaded standalone machine the parallel OpenMP map/reduce code with 2 thread runs 1.27 times faster than sequential one, and the speed-up scales up to 4 threads with maximal 1.6 relative speedup. Cilk Plus reducer with multi-threads, however, has poorer performance than sequential code, and its speedup drops down with number of threads.

9.5.4.2 Performance Evaluation on Virtual Machine

The parallel benchmarks of LZ77 OpenMP map/reduce and Cilk Plus reducer programs were performed on 8 core (16 hyper-threads) Google Compute Engine (GCE) (Intel(R) Xeon(R) CPU @ 2.20GHz and 16 GB memory). The C compilers in benchmarks include GCC 5.4, which has been integrated with

Intel Cilk Plus. The detailed benchmark results are list in Appendix Table C.4.

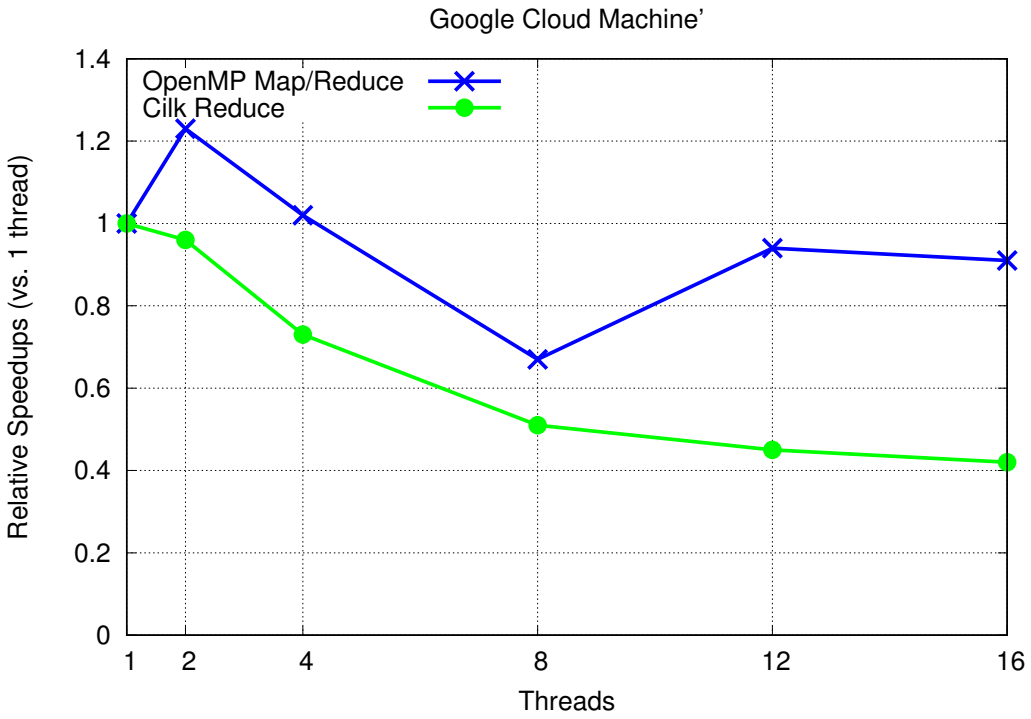


Figure 9.16: Relative speedup (vs. 1 Thread) of parallel LZ77 programs on 8-core (up to 16 threads) Google Compute Engine machine (Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz and 16 GB memory)

Figure 9.16 shows the benchmark results on 8-core Google Cloud machine. The OpenMP code outperforms Cilk Plus code in all kinds of threads and has a slightly better performance than sequential C code when two threads are used. However, as the number of thread increases, the OpenMP code does not scale up the speedup but slows down the execution.

Conclusion Polly compiler does not parallelise LZ77 compress program because the none-affine loop condition used in the algorithm requires more information for automatic parallelism. Both of OpenMP and Cilk Plus map/reducer programs are slower than Polly sequential code. The slow-down may result from expensive overheads of reducers. OpenMP program spends extra time creating and freeing local arrays, and Cilk Plus program has similar overheads but also include additional work to manage the queue for each thread.

9.6 Summary

Through these experiments and case studies, we learn that not every program can be parallelised to achieve good and scalable speed-ups, and the speed-up of the entire program is limited by the part that cannot benefit from parallelism, according to Amdahl's law (Mittal and Vetter, 2015). For example, our bubble sorting program does not exhibit any parallelism, whereas merge sort obtains the scalable performance from multi-processor execution.

The parallelisable loop must not have data dependency nor execution order, so that the computation of the loop can be carried out concurrently to gain speedups from out-of-order execution and avoid waiting time. We can use some techniques to analyse loop data dependency. For example, in coin game case we draw out an iteration space graph, where each node is one loop iteration and each path is the data flow from one node to another, to show whether the data dependency is carried out within loop iterations.

The parallelisable task can be partitioned into small sub-tasks and each sub-task can be computed separately and individually without needing any data from other sub-tasks. For example, in our LZ77 compression case we split the time-consuming match searching procedure to a number of sub-tasks. Each sub-task takes one part of input data and follows the same procedure to search the optimal result locally. Once all the sub-tasks finish, the reducer subsequently collects all the local optimal results from all sub-tasks, and merge to one global optimal result. By scheduling these sub-tasks on multiple processors and running them concurrently, the workloads can be shared to achieve load balancing and maximise the throughouts and minimise the waiting time.

However, the parallelising process, such as identifying the parallelisable loops and code transform, still heavily relies on human efforts. Besides, to gain portable speed-ups across various architectures requires the knowledge of performance tuning, but also lots of experiment efforts to find the optimal configuration setting. When the program becomes larger, these tasks become too complicated and tedious to be done by hand.

Thus, an effective paralleling framework is urgently needed to analyse a program and transform the sequential code into the parallel code in a systematised way. The framework would firstly detect the parallelisable part in the program and choose a proper and suitable parallel technique, such as map-reduce style, depending on the data dependency and data-flow controls. Then, the compiler converts the sequential program into parallel code whilst validating the safety of parallel code to avoid common multi-threaded problems, e.g. race conditions and deadlocks. Lastly, the performance tuning analyser runs some experiments on the parallel code and measure the performance to obtain its optimal configuration (e.g. task granularity), and to exploit the maximal parallelism on target machines and scale the performance up.

At the time of writing, we have not found any compiler or useful tool that can automatically parallelise the verification-friendly Whiley program to run on multiple CPUs and/or GPUs. Building such a parallelising compiler is one of our future work, although we know there are many challenges along the way.

Chapter 10

Conclusions and Future Work

The Whiley programming language employs extended static checking to eliminate run-time errors at compile time such that a Whiley program can be converted into different programming languages and executed correctly across a variety of run-time environments.

Our project builds up an optimising Whiley-to-C compiler to generate fast, memory-efficient and safe C code from a Whiley program. Our project is built around Whiley intermediate language (WyIL) code produced by the Whiley compiler and includes several *static* code analysers along with an *automatic* code generator.

Our pattern matcher and bound analyser enable the code generator to provide estimated integer intervals to make use of fixed-width integer types in translation, but the evaluation is not conducted in this work.

Our copy elimination and deallocation analysers can further improve the efficiency of generated C code by removing unnecessary array copies and memory leaks. Moreover, our combined static analysis, macro and run-time flags ensure every memory block is de-allocated exactly once and guarantee memory safety during program execution.

Semi-formal proofs are constructed by hand to verify all our deallocation macros do not free a memory block after it has been freed. To further validate our deallocation macros, we also used automatic theorem prover *Boogie* to

mechanically verify that each macro preserves our deallocation invariant.

Our Whiley-to-C compiler is used in 9 benchmark programs. Each Whiley benchmark program is automatically translated and optimised into sequential C code without manual interaction. The benchmark results show our optimal code runs at low overheads without expensive and unneeded array copies and effectively stops all memory leaks without violating memory safety at run-time. As such, the optimised code runs securely, fast and for long periods whilst maintaining the program correctness.

Future Work Our code generator supports the stable version (*v0.3.39*) of the Whiley programming language, and needs an upgrade to support new WyIL code types provided by a newer version of Whiley compiler (*v0.4.1*).

Our project targets the optimisation of one-dimensional array of primitive types (without cyclic references) in Whiley. The support of multi-dimensional array, recursive data type or any nested structure requires the re-design of deallocation responsibility. For example, a new design of size variable is needed to store the length of non-rectangular array, and a set of new multi-level macros is also needed to monitor the ownership of de-allocating sub-arrays at run-time.

Recursion will be supported as a part of the future work, by iterating the static analysis steps until convergence. For example, Tarjan's strongly connected components algorithm (Tarjan, 1972) used by `gprof` (Graham et al., 2004) can identify the mutually recursive functions, so that our analyser can use a special strategy to perform the analysis on these functions.

The Whiley verification features can be leveraged to improve the provision of our static optimisation. Pre- and post-conditions can be incorporated into our bound analyser to estimate the array values and sizes to decide the array variable types, particularly those variables whose ranges cannot be determined statically to fit into fix-sized data types, such as reading the value from a file. The loop invariant can also be useful for code optimisation, e.g. dynamic-growing arrays can be transformed to fixed-size arrays for lower overheads.

Our static copy and deallocation analyses can also benefit from Whiley verification specifications. For some uncertain situations where the parameter may or may not be returned, our analyser tends to keep the array copy and then remove the unneeded copy dynamically after the call. This conservative strategy incurs overheads at run-time. By specifying no aliasing information in the pre- and post-conditions or loop invariant, these unneeded array copies can be eliminated at compile time to reduce the run-time overheads.

Future work could include a compiler tool-kit to point out these questionable variables and the uncertainty in function calls, and give the suggestions to users to include extra assumptions or loop invariants in the Whiley programs. By doing so, program correctness can be ensured and the quality of generated code can be improved.

Our deallocation depends on the several static analyses to place the macros in the generated code and manage the memory deallocation at run-time. These analyses (live variable, return and mutability analysis) require further formalisation to verify their correctness. This future work can further strengthen the proofs in Section 6.

We experimented with fully automatic and semi-automatic parallelism on our optimised generated C code, and our results show that these parallel techniques have various impact on the performance, depending on the amount of available parallelism that can exploit in the program, and the computing resources that each technique provides. Future research could use the observations of our parallel experiments to develop a parallelisation heuristics to exploit the parallelism in Whiley, and build up an automatic parallelisation framework to select effective technique and scale up the performance across different platforms and devices.

References

Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*, chapter 8, pages 529–531. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986. ISBN 0-201-10088-6.

Jonathan Aldrich, Valentin Kostadinov, and Craig Chambers. Alias annotations for program understanding. In *ACM SIGPLAN Notices*, volume 37, pages 311–330. ACM, 2002.

Andrei Alexandrescu. *Modern C++ design: generic programming and design patterns applied*. Addison-Wesley, 2001.

Sean Eron Anderson. Bit twiddling hacks, 2005. URL `http://graphics.stanford.edu/~seander/bithacks.html#IntegerMinOrMax`. [Online; accessed 29-April-2017].

Mahwish Arif and Hans Vandierendonck. A case study of openmp applied to map/reduce-style computations. In *International Workshop on OpenMP*, pages 162–174. Springer, 2015.

Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérome Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A Static Analyzer for Large Safety-critical Software. *SIGPLAN Not.*, 38(5):196–207, May 2003. ISSN 0362-1340. doi: 10.1145/780822.781153. URL `http://doi.acm.org/10.1145/780822.781153`.

Jim Blandy. *Why Rust? Trustworthy, Concurrent Systems Programming*. O'Reilly, 2015.

Robert D Blumofe and Charles E Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)*, 46(5):720–748, 1999.

Victor Hugo Sperle Campos, Raphael Ernani Rodrigues, Igor Rafael de Assis Costa, and Fernando Magno Quinto Pereira. Speed and Precision in Range Analysis. In FranciscoHeron de Carvalho Junior and Luis-Soares Barbosa, editors, *Programming Languages*, volume 7554 of *Lecture Notes in Computer Science*, pages 42–56. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-33181-7. doi: 10.1007/978-3-642-33182-4_5. URL `http://dx.doi.org/10.1007/978-3-642-33182-4_5`.

Barbara Chapman, Gabriele Jost, and Ruud Van Der Pas. *Using OpenMP: portable shared memory parallel programming*, volume 10. MIT press, 2008.

Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. Vcc: A practical system for verifying concurrent c. In *International Conference on Theorem Proving in Higher Order Logics*, pages 23–42. Springer, 2009.

Thomas H Cormen. *Introduction to algorithms*. MIT press, 2009.

Agostino Cortesi and Matteo Zanioli. Widening and Narrowing Operators for Abstract Interpretation. *Computer Languages, Systems & Structures*, 37(1): 24–42, 2011.

Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-78800-3.

Peter J Denning. Thrashing: Its causes and prevention. In *Proceedings of the December 9-11, 1968, fall joint computer conference, part I*, pages 915–922. ACM, 1968.

Johannes Doerfert, Kevin Streit, Sebastian Hack, and Zino Benaissa. Polly's Polyhedral Scheduling in the Presence of Reductions. *arXiv preprint arXiv:1505.07716*, 2015.

Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended Static Checking for Java. *SIGPLAN Not.*, 37(5):234–245, May 2002. ISSN 0362-1340. doi: 10.1145/543552.512558. URL `http://doi.acm.org/10.1145/543552.512558`.

Matteo Frigo, Pablo Halpern, Charles E Leiserson, and Stephen Lewin-Berlin. Reducers and other cilk++ hyperobjects. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, pages 79–90. ACM, 2009.

Jdrzej Fulara and Krzysztof Jakubczyk. Practically Applicable Formal Methods. In Jan van Leeuwen, Anca Muscholl, David Peleg, Jaroslav Pokorn, and Bernhard Rumpe, editors, *SOFSEM 2010: Theory and Practice of Computer Science*, volume 5901 of *Lecture Notes in Computer Science*, pages 407–418. Springer Berlin Heidelberg, 2010. ISBN 978-3-642-11265-2. doi: 10.1007/978-3-642-11266-9_34. URL `http://dx.doi.org/10.1007/978-3-642-11266-9_34`.

Thomas Gawlitza, Jrme Leroux, Jan Reineke, Helmut Seidl, Grgoire Sutre, and Reinhard Wilhelm. Polynomial Precise Interval Analysis Revisited. In Susanne Albers, Helmut Alt, and Stefan Nher, editors, *Efficient Algorithms*, volume 5760 of *Lecture Notes in Computer Science*, pages 422–437. Springer Berlin Heidelberg, 2009. ISBN 978-3-642-03455-8. doi: 10.1007/978-3-642-03456-5_28. URL `http://dx.doi.org/10.1007/978-3-642-03456-5_28`.

K Gopinath and John L Hennessy. Copy elimination in functional languages. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 303–314. ACM, 1989.

Deepak Goyal and Robert Paige. A new solution to the hidden copy problem. In *International Static Analysis Symposium*, pages 327–348. Springer, 1998.

Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. Gprof: A Call Graph Execution Profiler. *SIGPLAN Not.*, 39(4):49–57, April 2004. ISSN 0362-1340. doi: 10.1145/989393.989401. URL `http://doi.acm.org/10.1145/989393.989401`.

Tobias Grosser, Armin Größlinger, and Christian Lengauer. PollyPerforming Polyhedral Optimizations on a Low-level Intermediate Representation. *Parallel Processing Letters*, 22(04):1250010, 2012.

Tobias Grosser, Sebastian Pop, J Ramanujam, and P Sadayappan. On Recovering Multi-dimensional Arrays in Polly. In *Proceedings of the Fifth International Workshop on Polyhedral Compilation Techniques (IMPACT)*, 2015a.

Tobias Grosser, Sven Verdoolaege, and Albert Cohen. Polyhedral AST Generation is more than Scanning Polyhedra. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 37(4):12, 2015b.

Sumit Gulwani, Sagar Jain, and Eric Koskinen. Control-flow Refinement and Progress Invariants for Bound Analysis. *SIGPLAN Not.*, 44(6):375–385, June 2009. ISSN 0362-1340. doi: 10.1145/1543135.1542518. URL `http://doi.acm.org/10.1145/1543135.1542518`.

Pablo Halpern. Strict fork-join parallelism. *WG21 paper N*, 3409, 2012.

Michael Hicks, Greg Morrisett, Dan Grossman, and Trevor Jim. Experience with safe manual memory-management in cyclone. In *Proceedings of the 4th international symposium on Memory management*, pages 73–84. ACM, 2004.

Tony Hoare. The Verifying Compiler: A Grand Challenge for Computing Research. *J. ACM*, 50(1):63–69, January 2003. ISSN 0004-5411. doi: 10.1145/602382.602403. URL `http://doi.acm.org/10.1145/602382.602403`.

Paul Hudak and Adrienne Bloss. The aggregate update problem in functional programming systems. In *Proceedings of the 12th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 300–314. ACM, 1985.

Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and reasoning about systems*. Cambridge university press, 2004.

Michael Kruse. *Lattice QCD Optimization and Polytopic Representations of Distributed Memory*. PhD thesis, Paris 11, 2014.

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.

Chris Lattner. Llvm and clang: Next generation compiler technology. In *The BSD Conference*, pages 1–2, 2008.

K. R. M. Leino. Accessible software verification with dafny. *IEEE Software*, 34(6):94–97, November/December 2017. ISSN 0740-7459. doi: 10.1109/MS.2017.4121212. URL `doi.ieeecomputersociety.org/10.1109/MS.2017.4121212`.

K Rustan M Leino. This is boogie 2. *Manuscript KRML*, 178(131), 2008.

K. Rustan M. Leino. Dafny: An Automatic Program Verifier for Functional Correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 348–370, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-17511-4.

Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. Compcert-a formally verified optimizing compiler. In *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*, 2016.

Petra Malik and Mark Utting. CZT: A Framework for Z Tools. In Helen Treharne, Steve King, Martin Henson, and Steve Schneider, editors, *ZB 2005: Formal Specification and Development in Z and B*, volume 3455 of *Lecture Notes in Computer Science*, pages 65–84. Springer Berlin Heidelberg, 2005. ISBN 978-3-540-25559-8. doi: 10.1007/11415787_5. URL `http://dx.doi.org/10.1007/11415787_5`.

Ovidio José Mallo. A translator from bml annotated java bytecode to boogiepl. *Software Component Technology Group, Department of Computer Science, ETH Zurich, Switzerland*, 2007.

Francesco Logozzo Manuel Fahndrich. Static contract checking with Abstract Interpretation. In *Proceedings of the Conference on Formal Verification of Object-oriented Software (FoVeOOS 2010)*. Springer Verlag, October 2010. URL `https://www.microsoft.com/en-us/research/publication/static-contract-checking-with-abstract-interpretation/`.

K. Marriott and P. J. Stuckey. *Programming with Constraints: An Introduction*. Adaptive Computation and Machine. MIT Press, 1998. ISBN 9780262133418. URL `http://books.google.co.nz/books?id=jBYAleHTldsC`.

Wolfram Schulte Mike Barnett, Rustan Leino. The Spec Programming System: An Overview. In *CASSIS 2004, Construction and Analysis of Safe, Secure and Interoperable Smart devices*, volume 3362, pages 49–69. Springer, January 2005. URL `https://www.microsoft.com/en-us/research/publication/the-spec-programming-system-an-overview/`.

Dmitry Mikushin and Nicolas Likhogrud. KernelGen — a Toolchain for Automatic GPU-centric Applications Porting. 2012.

Sparsh Mittal and Jeffrey S. Vetter. A survey of cpu-gpu heterogeneous computing techniques. *ACM Comput. Surv.*, 47(4):69:1–69:35, July 2015. ISSN

0360-0300. doi: 10.1145/2788396. URL http://doi.acm.org/10.1145/2788396.

Simon Moll, Johannes Doerfert, and Sebastian Hack. Input Space Splitting for OpenCL. 2016. http://compilers.cs.uni-saarland.de/papers/moll_pollocl.pdf.

Jorge A. Navas, Peter Schachte, Harald Sndergaard, and PeterJ. Stuckey. Signedness-Agnostic Program Analysis: Precise Integer Bounds for Low-Level Code. In Ranjit Jhala and Atsushi Igarashi, editors, *Programming Languages and Systems*, volume 7705 of *Lecture Notes in Computer Science*, pages 115–130. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-35181-5. doi: 10.1007/978-3-642-35182-2_9. URL http://dx.doi.org/10.1007/978-3-642-35182-2_9.

Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavy-weight dynamic binary instrumentation. In *ACM Sigplan notices*, volume 42, pages 89–100. ACM, 2007.

John K. Ousterhout, Ken Jones, Eric Foster-Johnson, Donal Fellows, Brian Griffin, and David Welton. *Tcl and the Tk Toolkit*. Addision-Wesley Professional Computing Series. Addison-Wesley, Upper Saddle River, New Jersey, 2 edition, 2010. ISBN 978-0-321-33633-0.

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

David 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, 2015a.

David J Pearce. *Practical verification condition generation for a bytecode lan-*

guage. School of Engineering and Computer Science, Victoria University of Wellington, 2015b.

David J Pearce and Lindsay Groves. Designing a verifying compiler: Lessons learned from developing whiley. *Science of Computer Programming*, 113: 191–220, 2015a.

David J. Pearce and Lindsay Groves. Designing a verifying compiler: Lessons learned from developing whiley. volume 113, pages 191 – 220. 2015b. doi: https://doi.org/10.1016/j.scico.2015.09.006. URL `http://www.sciencedirect.com/science/article/pii/S016764231500266X`. Formal Techniques for Safety-Critical Systems.

Jonathan Protzenko, Jean-Karim Zinzindohoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella-Béguelin, Antoine Delignat-Lavaud, Cătălin Hriţcu, Karthikeyan Bhargavan, Cédric Fournet, and Nikhil Swamy. Verified Low-level Programming Embedded in F*. *Proc. ACM Program. Lang.*, 1(ICFP):17:1–17:29, August 2017a. ISSN 2475-1421. doi: 10.1145/3110261. URL `http://doi.acm.org/10.1145/3110261`.

Jonathan Protzenko, Jean-Karim Zinzindohoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella-Béguelin, Antoine Delignat-Lavaud, Cătălin Hriţcu, Karthikeyan Bhargavan, Cédric Fournet, et al. Verified low-level programming embedded in f. *Proceedings of the ACM on Programming Languages*, 1(ICFP):17, 2017b.

A Raghesh. *A Framework for Automatic OpenMP Code Generation*. PhD thesis, Indian Institute of Technology, Madras, 2011.

Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Summer Deng, Roman Dzhabarov, James Hegeman, Roman Levenstein, Bert Maher, Satish Nadathur, Jakob Olesen, et al. Glow: Graph lowering compiler techniques for neural networks. *arXiv preprint arXiv:1805.00907*, 2018.

Peter Schnorf, Mahadevan Ganapathi, and John L Hennessy. Compile-time Copy Elimination. *Software: Practice and Experience*, 23(11):1175–1200, 1993.

Helmut Seidl, Reinhard Wilhelm, and Sebastian Hack. *Compiler Design: Analysis and Transformation*. Springer Science & Business Media, 2012.

Olha Shkaravska, Rody Kersten, and Marko van Eekelen. Test-based inference of polynomial loop-bound functions. In *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java*, PPPJ '10, pages 99–108, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0269-2. doi: 10.1145/1852761.1852776. URL `http://doi.acm.org/10.1145/1852761.1852776`.

Irvin Sobel. An isotropic 3× 3 image gradient operator. *Machine vision for three-dimensional scenes*, pages 376–379, 1990.

Richard M. Stallman and the GCC Developer Community. *Using the GNU Compiler Collection*. GNU Press, a division of the Free Software Foundation, 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA, 2003. URL `https://gcc.gnu.org/onlinedocs/gcc-5.2.0/gcc.pdf`. [Online; accessed 21-August-2017].

Zhendong Su and David Wagner. A class of polynomially solvable range constraints for interval analysis without widenings and narrowings. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2988 of *Lecture Notes in Computer Science*, pages 280–295. Springer Berlin Heidelberg, 2004. ISBN 978-3-540-21299-7. doi: 10.1007/978-3-540-24730-2_23. URL `http://dx.doi.org/10.1007/978-3-540-24730-2_23`.

Jim Sukha. A Quick Introduction to the Intel Cilk Plus Runtime, 2015. URL `https://www.cilkplus.org/sites/default/files/papers/CilkPlusRuntimeTutorial.pdf`.

Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.

LLVM Team. LLVM Language Reference Manual, 2016. URL `http://llvm.org/docs/LangRef.html`.

Rust team. Guide to rustc development. Technical report, Mozilla Research, 2019. URL `https://rust-lang.github.io/rustc-guide/`.

Julian Tschannen, Carlo A Furia, Martin Nordio, and Bertrand Meyer. Verifying eiffel programs with boogie. *arXiv preprint arXiv:1106.4700*, 2011.

Mark Utting, David J Pearce, and Lindsay Groves. Making whiley boogie! In *International Conference on Integrated Formal Methods*, pages 69–84. Springer, 2017.

Julien Vanegue and Shuvendu Lahiri. Towards practical reactive security audit using extended static checkers. In *IEEE Symposium on Security and Privacy (Oakland'13)*, May 2013. URL `https://www.microsoft.com/en-us/research/publication/towards-practical-reactive-security-audit-using-extended-static-checkers/`.

Mitchell Wand and William D Clinger. Set constraints for destructive array update optimization. *Journal of Functional Programming*, 11(3):319–346, 2001.

Min-Hsien Weng, Mark Utting, and Bernhard Pfahringer. Bound analysis for whiley programs. *Electronic Notes in Theoretical Computer Science*, 320: 53–67, 2016.

Min-Hsien Weng, Bernhard Pfahringer, and Mark Utting. Static techniques for reducing memory usage in the c implementation of whiley programs. In *ACSW'17*. ACM, 2017.

Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data

compression. *IEEE Transactions on information theory*, 23(3):337–343, 1977.

Appendix A

Boogie Program

Listing A.1: Deallocation macros Boogie program

```
0  type VAR; // Generic variable types
1  type AVAR; // Array variable
2  type ADDR; // Adrress variable
3  const unique null: ADDR;
4  var e: [AVAR]ADDR;// maps an array variable to its addresses.
5  var dealloc: [AVAR]bool;// a deallocation flag for each array variable
6  var valid: [ADDR]bool;// an address is valid if it has been heap−allocated,
       and not yet freed.
7  // define fresh(i) to describe if variable i is a fresh address
8  //function fresh(i: AVAR, e: [AVAR]ADDR) returns (r: bool);
9  //axiom (∀ i: AVAR, e: [AVAR]ADDR • fresh(i, e) ⟺ (∀ j: AVAR • i ≠ j
       ⟹ e[i] ≠ e[j]));
10 // define INV to describe deallocation invariant: inv_dealloc(i, j), inv_arr(i)
11 function INV(e: [AVAR]ADDR, dealloc: [AVAR]bool, valid: [ADDR]bool)
       returns (r: bool);
12 axiom (∀ e: [AVAR]ADDR, dealloc: [AVAR]bool, valid: [ADDR]bool•
13         INV(e, dealloc, valid)
14         ⟺ (∀ i,j: AVAR • dealloc[i] ∧ dealloc[j] ∧ i ≠ j ⟹ e[i
       ] ≠ e[j]) // inv_dealloc (i, j)
15               ∧ (∀ i: AVAR • dealloc[i] ⟹ valid[e[i]]) // inv_arr(i)
16         );
17
18 // Define free(a) to delete array 'a'
19 procedure freed(a: AVAR) returns ();
20      requires valid[e[a]];
21      modifies valid;
22      ensures valid[e[a]] = false;
23      ensures (∀ d: ADDR • d ≠ e[a] ⟹ valid[d] = old(valid[d]));
24
25 // Pre_dealloc Macro to free array if possible
26 procedure pre_dealloc(a: AVAR) returns ();
27      requires INV(e, dealloc, valid);
28      modifies e, dealloc, valid;
29      ensures INV(e, dealloc, valid);
30      ensures (∀ i: AVAR • i ≠ a ⟹ e[i] = old(e[i]));
31      ensures (∀ d: ADDR • d ≠ old(e[a]) ⟹ valid[d] = old(valid[d])
        );
32      ensures (∀ i: AVAR • i ≠ a ⟹ dealloc[i] = old(dealloc[i]));
33      ensures dealloc[a] = false;
34 implementation pre_dealloc(a: AVAR) returns ()
35 {
```

```
36      if (dealloc[a]) {
37          call freed(a); // free(a)
38          e := e[a := null];// e[a] := null;
39          dealloc := dealloc[a := false];// a_dealloc := false
40      }
41  }
42
43
44  // Create a new address 'r'
45  procedure malloc() returns (r: ADDR);
46      modifies valid;
47      ensures valid[r];
48      ensures (∀ i: AVAR • e[i] ≠ r); // fresh(r)
49      ensures (∀ d: ADDR • d ≠ r ⟹ valid[d] = old(valid[d]));
50
51  // New array
52  procedure new_array(a: AVAR) returns();
53      requires INV(e, dealloc, valid);
54      modifies e, dealloc, valid;
55      ensures valid[e[a]];
56      ensures dealloc[a];
57      ensures (∀ i: AVAR • i ≠ a ⟹ e[i] = old(e[i]));
58      ensures (∀ d: ADDR • d ≠ old(e[a]) ∧ d ≠ e[a] ⟹ valid[d] =
        old(valid[d]));
59      ensures (∀ i: AVAR • i ≠ a ⟹ dealloc[i] = old(dealloc[i]));
60      ensures INV(e, dealloc, valid);
61  implementation new_array(a: AVAR) returns ()
62  {
63      var ret: ADDR;
64      call pre_dealloc(a);
65      call ret := malloc();
66      e := e[a := ret]; // e[a] = e[ret]
67      dealloc := dealloc[a := true];
68  }
69
70  // define 'a := copy(b)' to make a copy of 'b' and return a fresh address 'a'
71  procedure copy(b: AVAR) returns (a: ADDR);   // Returns ADDR
72      requires valid[e[b]];
73      modifies valid;
74      ensures valid[a];
75      ensures (∀ i: AVAR • e[i] ≠ a); // fresh(a)
76      ensures (∀ d: ADDR • d ≠ a ⟹ valid[d] = old(valid[d]));
77
78  // Add_Dealloc Macro, a : = copy(b)
79  procedure add_dealloc(a: AVAR, b: AVAR) returns();
80      requires e[a] ≠ e[b] ∧ INV(e, dealloc, valid) ∧ valid[e[b]];
81      modifies e, dealloc, valid;
82      ensures INV(e, dealloc, valid);
83      ensures (∀ i: AVAR • i ≠ a ⟹ e[i] = old(e[i]));   // i ≠ old(a)
        should not be included ˜˜˜Robi
84      ensures (∀ d: ADDR • d ≠ old(e[a]) ∧ d ≠ e[a] ⟹ valid[d] =
        old(valid[d]));
85      ensures (∀ i: AVAR • i ≠ a ⟹ dealloc[i] = old(dealloc[i]));
86      ensures valid[e[a]] ∧ valid[e[b]] ∧ dealloc[a];
87  implementation add_dealloc(a: AVAR, b: AVAR) returns ()
88  {
89      var ret: ADDR; // Local variables
90      call pre_dealloc(a);
91      assert valid[e[b]];
92      call ret := copy(b);
93      e := e[a := ret]; // e[a] = e[ret]
94      dealloc := dealloc[a := true];
95  }
96
97  // Transfer_Dealloc Macro, a : = b
```

```
98  procedure transfer_dealloc(a: AVAR, b: AVAR) returns();
99      requires e[a] ≠ e[b] ∧ INV(e, dealloc, valid) ∧ valid[e[b]];
100     modifies e, dealloc, valid;
101     ensures INV(e, dealloc, valid);
102     ensures (∀ i: AVAR • i ≠ a ⟹ e[i] = old(e[i]));
103     ensures (∀ d: ADDR • d ≠ old(e[a]) ⟹ valid[d] = old(valid[d])
        );
104     ensures (∀ i: AVAR • i ≠ a ∧ i ≠ b ⟹ dealloc[i] = old(
        dealloc[i]));
105     ensures valid[e[a]] ∧ dealloc[a] = old(dealloc[b]);
106     ensures e[a] = e[b];
107     ensures ¬dealloc[b];// also ensures ¬dealloc[b]
108 implementation transfer_dealloc(a: AVAR, b: AVAR) returns ()
109 {
110     var ret: ADDR; // Local variables
111     call pre_dealloc(a);
112     assert valid[e[b]];
113     e[a] := e[b];
114     dealloc := dealloc[a := dealloc[b]];
115     dealloc := dealloc[b := false];
116 }
117
118 // Function func does not change array 'b', but returns a new array 'a'
119 procedure retain_func(b: AVAR, flag: bool) returns (r: ADDR);
120     requires valid[e[b]];
121     requires ¬flag;
122     modifies valid;
123     ensures valid[r];
124     ensures (∀ i: AVAR • e[i] ≠ r); // fresh(r)
125     ensures (∀ d: ADDR • d ≠ r ⟹ valid[d] = old(valid[d]));
126
127 // Retain_Dealloc Macro
128 procedure retain_dealloc(a: AVAR, b: AVAR) returns();
129     requires e[a] ≠ e[b] ∧ INV(e, dealloc, valid) ∧ valid[e[b]];
130     modifies e, dealloc, valid;
131     ensures (∀ i: AVAR • i ≠ a ⟹ e[i] = old(e[i]));
132     ensures (∀ d: ADDR • d ≠ old(e[a]) ∧ d ≠ e[a] ⟹ valid[d] =
        old(valid[d]));
133     ensures (∀ i: AVAR • i ≠ a ⟹ dealloc[i] = old(dealloc[i]));
134     ensures valid[e[a]];
135     ensures valid[e[b]];
136     ensures dealloc[a];
137     ensures INV(e, dealloc, valid);
138 implementation retain_dealloc(a: AVAR, b: AVAR) returns ()
139 {
140     var ret:  ADDR; // Local variable
141     call pre_dealloc(a); // pre_dealloc(a);
142     call ret := retain_func(b, false); // ret:=func(b, false);
143     e := e[a := ret]; // a:=ret;
144     dealloc := dealloc[a := true]; //a_dealloc := true
145 }
146
147
148 // Function func does not change array b, but may or may not returns array b
149 // This function is shared by reset and caller macros
150 procedure reset_caller_func(b: AVAR, flag: bool) returns (r: ADDR);
151     requires valid[e[b]];
152     requires ¬flag;
153     modifies valid;
154     ensures valid[r];
155     ensures ((∀ i: AVAR • e[i] ≠ r) ∨ (r = e[b])); // fresh(r) or r =
        e(b)
156     ensures (∀ d: ADDR • d ≠ r ⟹ valid[d] = old(valid[d]));
157
158 // Reset_Dealloc Macro
```

```
159 procedure reset_dealloc(a: AVAR, b: AVAR) returns();
160     requires e[a] ≠ e[b] ∧ INV(e, dealloc, valid) ∧ valid[e[b]];
161     modifies e, dealloc, valid;
162     ensures (∀ i: AVAR • i ≠ a ⟹ e[i] = old(e[i]));
163     ensures (∀ d: ADDR • d ≠ old(e[a]) ∧ d ≠ e[a] ⟹ valid[d] =
        old(valid[d]));
164     ensures (∀ i: AVAR • i ≠ a ∧ i ≠ b ⟹ dealloc[i] = old(
        dealloc[i]));// a_dealloc and b_dealloc
165     ensures valid[e[a]];
166     ensures valid[e[b]];
167     ensures INV(e, dealloc, valid);
168 implementation reset_dealloc(a: AVAR, b: AVAR) returns ()
169 {
170     var ret: ADDR; // local variables
171     call pre_dealloc(a); // pre_dealloc(a);
172     assert valid[e[b]];
173     call ret := reset_caller_func(b, false); // ret:=func(b, false);
174     e := e[a := ret]; // a:=ret;
175     assert valid[e[a]];
176     if(e[a] ≠ e[b]){
177         dealloc := dealloc[a := true]; //a_dealloc := true
178     }else{
179         dealloc := dealloc[a := dealloc[b]];
180         dealloc := dealloc[b := false];
181     }
182 }
183
184 // Caller_Dealloc Macro
185 procedure caller_dealloc(a: AVAR, b: AVAR, tmp: AVAR) returns();
186     requires tmp ≠ a;//Without this precondition, we can not prove the
        validity of all address
187     // E.g. valid(old(tmp)) is false. After caller Macro, we get valid(tmp) is
        true.
188     // So it might break the validity invariant. Therefore, the termination of
        Boogie is not guarantteed.
189     requires e[a] ≠ e[b] ∧ INV(e, dealloc, valid) ∧ valid[e[b]];
190     modifies e, dealloc, valid;
191     ensures (∀ i: AVAR • i ≠ a ∧ i ≠ tmp ⟹ e[i] = old(e[i]));
192     ensures (∀ d: ADDR • d ≠ old(e[a]) ∧ d ≠ e[a] ∧ d ≠ e[tmp] ⟹
         valid[d] = old(valid[d])); // validity invariant
193     ensures (∀ i: AVAR • i ≠ a ∧ i ≠ tmp ⟹ dealloc[i] = old(
        dealloc[i]));
194     ensures valid[e[a]];
195     ensures dealloc[a];
196     ensures INV(e, dealloc, valid);
197 implementation caller_dealloc(a: AVAR, b: AVAR, tmp: AVAR) returns ()
198 {
199     var ret: ADDR;
200     assume {: captureState "top"} true;// Capture intermediate states in
        the procedure body
201     call pre_dealloc(a); // pre_dealloc(a)
202     call ret := copy(b);
203     e := e[tmp := ret]; // tmp:= ret
204     dealloc := dealloc[tmp := false]; //tmp_dealloc := false
205     call ret := reset_caller_func(tmp, false); // ret:=func(b, false);
206     e := e[a := ret]; // a:=ret;
207     //assert a ≠ tmp;
208     if(e[a] ≠ e[tmp]){
209         call freed(tmp);
210     }
211     dealloc := dealloc[a := true]; //a_dealloc := true
212 }
213
214 // Function func may change array tmp but does not return array tmp
215 procedure callee_func(tmp: AVAR, flag: bool) returns (r: ADDR);
```

```
216     requires valid[e[tmp]];
217     requires flag;
218     modifies valid;
219     ensures ¬valid[e[tmp]]; // Free 'tmp'
220     ensures valid[r]; // valid address
221     ensures (∀ i: AVAR • e[i] ≠ r); // fresh(r)
222     ensures (∀ d: ADDR • d ≠ r ∧ d ≠ e[tmp] ⟹ valid[d] = old(
        valid[d]));
223
224 // Callee_Dealloc Macro
225 procedure callee_dealloc(a: AVAR, b: AVAR, tmp: AVAR) returns();
226     requires a ≠ tmp; // a and tmp are different variables
227     requires e[a] ≠ e[b] ∧ INV(e, dealloc, valid) ∧ valid[e[b]];
228     modifies e, dealloc, valid;
229     ensures (∀ i: AVAR • i ≠ a ∧ i ≠ tmp ⟹ e[i] = old(e[i]));
230     ensures (∀ d: ADDR • d ≠ old(e[a]) ∧ d ≠ e[a] ∧ d ≠ e[tmp] ⟹
        valid[d] = old(valid[d]));
231     ensures (∀ i: AVAR • i ≠ a ∧ i ≠ tmp ⟹ dealloc[i] = old(
        dealloc[i]));
232     ensures valid[e[a]];
233     ensures ¬valid[e[tmp]];
234     ensures dealloc[a];
235     ensures INV(e, dealloc, valid);
236 implementation callee_dealloc(a: AVAR, b: AVAR, tmp: AVAR) returns ()
237 {
238     var ret: ADDR;
239     call pre_dealloc(a); // pre_dealloc(a)
240     call ret := copy(b);
241     e := e[tmp := ret]; // tmp := ret;
242     dealloc := dealloc[tmp := true]; //tmp_dealloc := true
243     //assert tmp ≠ b;
244     dealloc := dealloc[tmp := true]; //tmp_dealloc := true
245     call ret := callee_func(tmp, true); // ret:=func(b, true);
246     e := e[a := ret]; // a:=ret;
247     //assert tmp ≠ a;
248     dealloc := dealloc[tmp := false]; //tmp_dealloc := false
249     dealloc := dealloc[a := true]; //a_dealloc := true
250 }
```

Appendix B

Sequential Benchmarks

B.1 Benchmark Whiley Program

Listing B.1: Reverse Whiley program

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

//public export method test() -> void:
method main(System.Console sys):
    int|null n = Int.parse(sys.args[0])
    if n != null:
        int max = n
        int size = 10000000
        int repeats = 0
        while repeats < max:
            //Reverse an array 'arr' ([max ... 0])
            int index = 0
            int[] arr = [0;size]
            //Fill in the array in the reverse order (10000000..0)
            while index < size:
                arr[index] = size - index
                index = index + 1
            //Sort the array
            arr = reverse(arr)
            /**Print the last element of sorted array */
            sys.out.println(arr[size-1])
            /** Print out the successful message */
            repeats = repeats + 1
        sys.out.print_s("Number of repeats: ")
        sys.out.println(repeats)
        sys.out.println_s("Pass Reverse test case ")
```

Listing B.2: TicTacToe Whiley program

```
import whiley.lang.*

type nat is (int x) where x >= 0

constant BLANK is 0
constant CIRCLE is 1
constant CROSS is 2

// A square is either blank, or a circle or cross.
type Square is (int x)
where x == BLANK || x == CIRCLE || x == CROSS

// A board consists of 9 squares, and a move counter
type Board is (null |{
    nat move,
    Square[] pieces // 3 x 3
} this)
where this != null &&
    |this.pieces| == 9 && this.move <= 9
where this != null &&
    countOf(this.pieces,BLANK) == (9 - this.move)
where this != null &&
    (countOf(this.pieces,CIRCLE) == countOf(this.pieces,CROSS) ||
    countOf(this.pieces,CIRCLE) == countOf(this.pieces,CROSS)+1)

// An empty board is one where all pieces are blank
function EmptyBoard() -> (Board r)
ensures r != null && r.move == 0:// Empty board has no moves yet
    return {
        move: 0,
        pieces: [BLANK,BLANK,BLANK,
                 BLANK,BLANK,BLANK,
                 BLANK,BLANK,BLANK]
    }

// Helper Method
function countOf(Square[] pieces, Square s) -> (int r):
    int count = 0
    int i = 0
    while i < |pieces|:
        if pieces[i] == s:
            count = count + 1
        i = i + 1
    return count

// Test Game
constant GAME is [0,1,2,3,4,5,6,7,8]

method main(System.Console sys):
    int|null n = Int.parse(sys.args[0])
    if n != null:
        int max = n
        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:
```

```
64                          b1.pieces[p]=CIRCLE
65                          b1.move = b1.move + 1
66                          b2 = b1
67                          b1 = null
68                      else:
69                          if b2 != null:
70                              b2.pieces[p]=CROSS
71                              b2.move = b2.move + 1
72                              // Move board to next player
73                              b1 = b2
74                              b2 = null
75                  i = i + 1
76              repeat = repeat + 1
77          sys.out.println_s("Pass newTicTacToe test case")
```

Listing B.3: Bubble sort Whiley program

```
import whiley.lang.*

function bubbleSort(int[] items) -> int[]:
  int length = |items|
  // The index of last swapped item.
  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 the remaing items as they are orderred.
    // This saves lots of time.
    length = last_swapped
  return items

method main(System.Console sys):
  int|null n = Int.parse(sys.args[0])
  if n != null:
    int max = n
    int size = 10000
    int repeats = 0
    while repeats < max:
          //Create a reverse array 'arr' ([10000 ... 1])
          int index = 0
          int[] arr = [0;size]
          //sys.out.println(arr)
          //Fill in the array in the reverse order (10000..1)
          while index < size:
                arr[index] = size - index
                index = index + 1
          //Sort the array
          arr = bubbleSort(arr)
          // Print the last element of sorted array
          //sys.out.println(arr[0])
          sys.out.println(arr[size-1])
          repeats = repeats + 1
    sys.out.print_s("Number␣of␣repeats␣")
    sys.out.println(repeats)
    sys.out.println_s("Pass␣BubbleSort␣test␣case")
```

Listing B.4: Merge sort Whiley program

```
import whiley.lang.*
// Perform a merge sort on integer array
function sortV1(int[] items, int start, int end)->int[]:
    if (start+1) < end:
        // First, split unsorted items into left and right sub-arrays
        int pivot = (start+end) / 2
        int[] lhs = Array.slice(items,start,pivot)
        lhs = sortV1(lhs, 0, pivot)// Recursively split left sub-array
        int[] rhs = Array.slice(items,pivot,end)
        rhs = sortV1(rhs, 0, (end-pivot))// Split right sub-array
        // Second, merge left and right sub-arrays into output array.
        int l = 0    // Starting index of left sub-array
        int r = 0    // Starting index of right sub-array
        int i = 0    // Starting index of output array
        // Update output array with smaller item of left and right sub-arrays
        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
        while l < (pivot-start):// Tidy up left sub-array
            items[i] = lhs[l]
            i=i+1
            l=l+1
        while r < (end-pivot):// Tidy up right sub-array
            items[i] = rhs[r]
            i=i+1
            r=r+1
        // Done
    return items

method main(System.Console sys):
    int|null n = Int.parse(sys.args[0])
    if n != null:
        int max = n
        int repeats = 0
        while repeats < max:
            // Create a reverse array
            int size = 10000
            int index = 0
            int[] arr = [0;size]
            //Fill in the array in the reverse order (1000..1)
            while index < size:
                arr[index] = size - index
                index = index + 1
            //Use merge sort to order reversed array 'arr' ([1000 ... 1])
            arr = sortV1(arr, 0, max)
            // Should be in the ascending order [1..1000]
            //sys.out.println(arr[0])
            sys.out.println(arr[max-1])
            repeats = repeats + 1
            sys.out.print_s("Number␣of␣repeats␣")
            sys.out.println(repeats)
        sys.out.println_s("Pass␣Mergesort␣test␣case")
```

Listing B.5: Matrix multiplication Whiley program

```
import whiley.lang.*
import whiley.io.File

// Initialize a Matrix
function init(int[] data, int width, int height) -> (int[] r):
    // Fill in Matrix
    int i = 0
    while i < height:
        int j = 0
        while j < width:
            data[i*width+j] = i
            j = j + 1
        i = i + 1
    return data

// Initialize a Matrix and assign each element with its row
function mat_mult(int[] a, int[] b, int[] data, int width, int height)
    -> (int[] c):
    int i = 0
    while i < height:
        int j = 0
        while j < width:
            int k = 0
            int sub_total = 0
            while k < width:
                // c[i][j] = c[i][j] + a[i][k] * b[k][j]
                sub_total = sub_total + a[i*width+k]*b[k*width+j]
                k = k + 1
            data[i*width+j] = sub_total
            j = j + 1
        i = i + 1
    return data

method main(System.Console sys):
    int|null n = Int.parse(sys.args[0])
    if n != null:
        int size = n
        int width = size
        int height = size
        sys.out.print_s("size␣=␣")
        sys.out.println(size)
        // Initialize matrix A
        int[] A = [0;width*height]
        A = init(A, width, height)
        // Initialize matrix B
        int[] B = [0;width*height]
        B = init(B, width, height)
        int[] C = [0;width*height]
        C = mat_mult(A, B, C, width, height)
        //sys.out.print_s("Matrix C[size-1][size-1] = ")
        sys.out.println(C[(size-1)*size+size-1])
        sys.out.println_s("Pass␣MatrixMult␣test␣case")
```

Listing B.6: Cash till Whiley program

```
import whiley.lang.*
/*
* The source code is from cashtill of Whiley benchmark suite
* https://github.com/Whiley/WyBench/blob/master/src/015_cashtill/Main.
    whiley
*/
type nat is (int n) where n >= 0

/**
 * Define coins/notes and their values (in cents)
 */
constant ONE_CENT is 0
constant FIVE_CENTS is 1
constant TEN_CENTS is 2
constant TWENTY_CENTS is 3
constant FIFTY_CENTS is 4
constant ONE_DOLLAR is 5   // 1 dollar
constant FIVE_DOLLARS is 6   // 5 dollars
constant TEN_DOLLARS is 7  // 10 dollars

constant Value is [
    1,
    5,
    10,
    20,
    50,
    100,
    500,
    1000
]

/**
 * Define the notion of cash as an array of coins / notes
 */
type Cash is (nat[] ns) where |ns| == |Value|

function Cash() -> Cash:
    return [0,0,0,0,0,0,0,0]

function Cash(nat[] coins) -> Cash
// No coin in coins larger than permitted values
requires all { i in 0..|coins| | coins[i] < |Value| }:
    Cash cash = [0,0,0,0,0,0,0,0]
    int i = 0
    while i < |coins|
        where |cash| == |Value|
                && all {k in 0..|cash| | cash[k] >= 0}:
        nat coin = coins[i]
        cash[coin] = cash[coin] + 1
        i = i + 1
    return cash

/**
 * Given some cash, compute its total
 */
function total(Cash c) -> int:
    int r = 0
    int i = 0
    while i < |c|:
        r = r + (Value[i] * c[i])
        i = i + 1
    return r
```

```
62
63  /**
64   * Checks that a second load of cash is stored entirely within the first.
65   * In other words, if we remove the second from the first then we do not
66   * get any negative amounts.
67   */
68  function contained(Cash first, Cash second) -> bool:
69      int i = 0
70      while i < |first|:
71          if first[i] < second[i]:
72              return false
73          i = i + 1
74      return true
75
76  /**
77   * Adds two bits of cash together
78   *
79   * ENSURES: the total returned equals total of first plus
80   *          the total of the second.
81   */
82  function add(Cash first, Cash second) -> (Cash r)
83  // Result total must be sum of argument totals
84  ensures total(r) == total(first) + total(second):
85      //
86      int i = 0
87      while i < |first|:
88          first[i] = first[i] + second[i]
89          i = i + 1
90      //
91      return first
92
93  /**
94   * Subtracts from first bit of cash a second bit of cash.
95   *
96   * REQUIRES: second cash is contained in first.
97   *
98   * ENSURES: the total returned equals total of first less
99   *          the total of the second.
100  */
101 function subtract(Cash first, Cash second) -> (Cash r)
102 // First argument must contain second; for example, if we have 1
103 // dollar coin and a 1 cent coin, we cannot subtract a 5 dollar note!
104 requires contained(first,second)
105 // Total returned must total of first argument less second
106 ensures total(r) == total(first) - total(second):
107     //
108     int i = 0
109     while i < |first|:
110         first[i] = first[i] - second[i]
111         i = i + 1
112     //
113     return first
114
115 /**
116  * Determine the change to be returned to a customer from a given cash
117  * till, assuming a certain cost for the item and the cash that was
118  * actually given.  Observe that the specification for this method does
119  * not dictate how the change is to be computed --- only that it must
120  * have certain properties.  Finally, if exact change cannot be given
121  * from the till then null is returned.
122  *
123  * ENSURES:  if change returned, then it must be contained in till, and
124  *           the amount returned must equal the amount requested.
125  */
126 function calculateChange(Cash till, nat change) -> (null|Cash r)
```

```
127 // If change is given, then it must have been in the till, and must equal that
        requested.
128 ensures r is Cash ==> (contained(till,r) && total(r) == change):
129     if change == 0:
130         return Cash()
131     else:
132         // exhaustive search through all possible coins
133         nat i = 0
134         while i < |till|:
135             if till[i] > 0 && Value[i] <= change:
136                 Cash tmp = till
137                 // temporarily take coin out of till
138                 tmp[i] = tmp[i] - 1
139                 null|Cash chg = calculateChange(tmp,
140                                                 change-Value[i])
141                 if chg != null:
142                     // we have enough change
143                     chg[i] = chg[i] + 1
144                     return chg
145             i = i + 1
146         return null// cannot give exact change :(
147 /**
148  * Print out cash in a friendly format
149  */
150 function toString(Cash c) -> ASCII.string:
151     ASCII.string r = ""
152     bool firstTime = true
153     int i = 0
154     while i < |c|:
155         int amt = c[i]
156         if amt != 0:
157             if !firstTime:
158                 r = Array.append(r,",␣")
159             firstTime = false
160             r = Array.append(r,Int.toString(amt))
161             r = Array.append(r,"␣x␣")
162             r = Array.append(r,Descriptions[i])
163         i = i + 1
164     if r == "":
165         r = "(nothing)"
166     return r
167
168 constant Descriptions is [
169     "␣1c",
170     "␣5c",
171     "10c",
172     "20c",
173     "50c",
174     "␣$1",
175     "␣$5",
176     "$10"
177 ]
178 /**
179  * Run through the sequence of a customer attempting to purchase an item
180  * of a specified cost using a given amount of cash and a current till.
181  */
182  public method buy(System.Console console, Cash till, Cash given,
        int cost) -> Cash:
183      if total(given) >= cost:
184         Cash|null change = calculateChange(till,total(given) - cost
    )
185         if change != null:
186              till = add(till,given)
187              till = subtract(till,change)
188      return till
```

```
189
190 /**
191  * Test Harness
192  */
193 public method main(System.Console console):
194     int|null n = Int.parse(console.args[0])
195     if n != null:
196         int max = n
197         int repeat = 0
198         while repeat < max:
199             // A cashtill is initialized with an empty array
200             Cash till = Cash()
201             // Change till every 2 iterations to avoid the same results
202             if repeat%2==1:
203                 // Initialize till with an empty array
204                 till = [5,3,3,1,1,3,0,0]
205             // console.out.print_s("Till: ")
206             // console.out.println_s(toString(till))
207             // now, run through some sequences...
208             till = buy(console,till,Cash([ONE_DOLLAR]),85)
209             till = buy(console,till,Cash([ONE_DOLLAR]),105)
210             till = buy(console,till,Cash([TEN_DOLLARS]),5)
211             till = buy(console,till,Cash([FIVE_DOLLARS]),305)
212             // console.out.print_s("Till: ")
213             // console.out.println_s(toString(till))
214             repeat = repeat + 1
```

Listing B.7: Coin game Whiley program

```
import whiley.lang.*
import whiley.io.File
import whiley.lang.Math

// Use dynamic programming to find moves for Alice
// The coins are an array, starting from 0 upto 5
function findMoves(int[] moves, int n, int[] coins) -> int[]:
  int s = 0
  while s < n: // 0<= s < n
    int i = 0
    while i < n -s: // 0 <= i < n -s
      int j = i + s // j = i + s
      int y = moves[(i + 1)*n+j - 1]
      int x = moves[(i + 2)*n+j]
      int z = moves[i*n+j - 2]
      moves[i*n+j] = Math.max(coins[i] + Math.min(x, y),
                              coins[j] + Math.min(y, z))
      i = i + 1
      // End of i,j loop
    s = s + 1
    // End of s loop
  return moves

method main(System.Console sys):
  int|null max = Int.parse(sys.args[0])
  if max != null:
     int n = max
     // Create an array of coins [0,1,2,3,4,0,1,2,3,4...]
     int[] coins = [0;n]
     int i = 0
     while i < n:
     coins[i] = i % 5 // Coin value [0 ~ 4]
     i = i + 1
     // Increase the move array size to (n+2) * (n+2)
     // to avoid if/else check inside the loop
     int[] moves = [0;(n+2)*(n+2)]
     moves = findMoves(moves, n, coins) // Pass 'moves' and 'coints'
    array
     //play(sys, moves, n)
     int sum_alice = moves[n-1]
     sys.out.print_s("Alice gets ")
     sys.out.println(sum_alice)
     sys.out.println_s("Pass CoinGame test case")
```

Listing B.8: LZ77 compression Whiley program

```
import * from whiley.io.File
import * from whiley.lang.System
import whiley.lang.*

// Positive integer type
type nat is (int x) where x >= 0
// Match type
type Match is ({nat offset, nat len} this)

// Find the matched entry with affine loop bound
function match(byte[] data, nat offset, nat end) -> (int length)
    ensures 0 <= length && length <= 255:
    nat pos = end
    nat len = 0
    while offset < pos && pos < |data| && data[offset] == data[pos]
     && len < 255:
        offset = offset + 1
        pos = pos + 1
        len = len + 1
    return len

// pos is current position in input value
function findLongestMatch(byte[] data, nat pos) -> (Match m):
    // Get 'data' byte array
    nat bestOffset = 0
    nat bestLen = 0
    int start = Math.max(pos - 255, 0)
    //assert start >= 0
    nat offset = start
    while offset < pos:
        int len = match(data, offset, pos)
        if len > bestLen:
            bestOffset = pos - offset
            bestLen = len
        offset = offset + 1
    // Return a 'Match' object
    return {offset:bestOffset, len:bestLen}

// Append a byte to the byte array
function append(byte[] items, byte item) -> (byte[] nitems):
    //
    nitems = [0b; |items| + 1]
    int i = 0
    //
    while i < |items|:
        nitems[i] = items[i]
        i = i + 1
    //
    nitems[i] = item
    return nitems

// Resize the input array to the array of given array size
function resize(byte[] items, int size) -> (byte[] nitems)
requires |items| >= size
ensures |nitems| == size:
    nitems = [0b; size]
    int i = 0
    while i < size:
        nitems[i] = items[i]
        i = i + 1
    //
    return nitems

```

```
63  // Compress 'input' array into 'output' array
64  function compress(byte[] data) -> (byte[] output):
65      nat pos = 0
66      // Initialize the output array of bytes
67      output = [0b;0]
68      // Iterate each byte in 'data'
69      while pos < |data|:
70          Match m = findLongestMatch(data, pos)
71          // Encode the match to 'offset-length' pair
72          // The distance to the longest match
73          byte offset = Int.toUnsignedByte(m.offset)
74          // The length of the match
75          byte length = Int.toUnsignedByte(m.len)
76          if offset == 00000000b:
77              // No match is found. Put the first byte of look-ahead array
78              length = data[pos]
79              pos = pos + 1
80          else:
81              // Skip the matched bytes
82              pos = pos + m.len
83          // Write 'offset-length' pair to the output array
84          output = append(output, offset)
85          output = append(output, length)
86      return output
87
88  // Decompress 'input' array to a string
89  function decompress(byte[] data) -> (byte[] output):
90      output = [0b;0]
91      nat pos = 0
92      //
93      while (pos+1) < |data|:
94          byte header = data[pos]
95          byte item = data[pos+1]
96          pos = pos + 2
97          if header == 00000000b:
98              output = append(output, item)
99          else:
100             int offset = Byte.toUnsignedInt(header)
101             int len = Byte.toUnsignedInt(item)
102             int start = |output| - offset
103             int i = start
104             while i < (start+len):
105                 // Get byte from output array
106                 item = output[i]
107                 //sys.out.println(item)
108                 output = append(output, item)
109                 i = i + 1
110     // all done!
111     return output
112
113 method main(System.Console sys):
114     // Read a text file of repeated contents as a byte array
115     File.Reader file = File.Reader(sys.args[0])
116     byte[] data = file.readAll()
117     sys.out.println_s("Data:␣␣␣␣␣␣␣␣␣")
118     sys.out.print(|data|)
119     sys.out.println_s("␣bytes")
120     // Compress the data with LZ algorithm
121     byte[] compress_data = compress(data)
122     sys.out.println_s("COMPRESSED␣Data:␣␣␣")
123     sys.out.print(|compress_data|)
124     sys.out.println_s("␣bytes")
```

Listing B.9: LZ77 decompression Whiley program using append array

```
/**
 * Simplified Lempel–Ziv 77 decompression.
 * See: http://en.wikipedia.org/wiki/LZ77_and_LZ78
 *      https://github.com/Whiley/WyBench/blob/master/src/009_lz77/Main.whiley
 */
import * from whiley.io.File
import * from whiley.lang.System
import whiley.lang.*

// Positive integer type
type nat is (int x) where x >= 0
// Append one byte to the array
function append(byte[] items, byte item) -> (byte[] nitems):
    nitems = [0b; |items| + 1]
    int i = 0
    //
    while i < |items|:
        nitems[i] = items[i]
        i = i + 1
    //
    nitems[i] = item
    return nitems
// Decompress 'input' array to a string
function decompress(byte[] data) -> (byte[] output):
    output = [0b;0]
    nat pos = 0
    //
    while (pos+1) < |data|:
        byte header = data[pos]
        byte item = data[pos+1]
        pos = pos + 2
        if header == 00000000b:
            output = append(output, item)
        else:
            int offset = Byte.toUnsignedInt(header)
            int len = Byte.toUnsignedInt(item)
            int start = |output| - offset
            int i = start
            while i < (start+len):
                // Get byte from output array
                item = output[i]
                //sys.out.println(item)
                output = append(output, item)
                i = i + 1
    // all done!
    return output

method main(System.Console sys):
    // Read the compress_data from a file
    File.Reader file = File.Reader(sys.args[0])
    byte[] input_data = file.readAll()
    // Decompress the data to a string
    byte[] decompress_data = decompress(input_data)
    sys.out.println_s("DECOMPRESSED:␣␣␣")
    sys.out.print(|decompress_data|)
    sys.out.println_s("␣bytes")
    file.close()
```

Listing B.10: LZ77 decompression Whiley Program using array list

```
/**
 * Lempel–Ziv 77 decompression using array list
 */
import * from whiley.io.File
import * from whiley.lang.System
import whiley.lang.*

type nat is (int x) where x >= 0// Positive integer type

// Resize the input array to the array of given array size
function resize(byte[] items, int size) -> (byte[] nitems)
requires |items| >= size
ensures |nitems| == size:
    nitems = [0b; size]
    int i = 0
    while i < size:
        nitems[i] = items[i]
        i = i + 1
    return nitems

// If full, then double array size and store the data
function opt_append(byte[] items, nat items_length, byte item) ->
    byte[]:
    if items_length < |items|:
        // Update the array without an array
        items[items_length] = item
    else:
        // Create a new array
        byte[] nitems = [0b; |items|*2+1]
        int i = 0
        while i < |items|:
            nitems[i] = items[i]
            i = i + 1
        nitems[i] = item
        items = nitems
    return items

// Decompress 'data' array to a byte array by using array list
function decompress(byte[] data) -> (byte[] output):
    byte[] items = [0b;0]
    nat items_length = 0
    nat pos = 0
    while (pos+1) < |data|:
        byte header = data[pos]
        byte item = data[pos+1]
        pos = pos + 2
        if header == 00000000b:
            items = opt_append(items, items_length, item)
            items_length = items_length + 1
        else:
            int offset = Byte.toUnsignedInt(header)
            int len = Byte.toUnsignedInt(item)
            int start = items_length - offset
            int i = start
            while i < (start+len):
                item = items[i]
                items = opt_append(items, items_length, item)
                items_length = items_length + 1
                i = i + 1
    //Resize list array into the array of accurate length
    output = resize(items, items_length)
    return output

```

```
method main(System.Console sys):
    // Read the compress_data from a file
    File.Reader file = File.Reader(sys.args[0])
    byte[] input_data = file.readAll()
    // Decompress the data to a string
    byte[] decompress_data = decompress(input_data)
    sys.out.println_s("DECOMPRESSED:␣␣␣␣")
    sys.out.print(|decompress_data|)
    sys.out.println_s("␣bytes")
    file.close()
```

Listing B.11: Sobel edge Whiley program

```
import * from whiley.io.File
import * from whiley.lang.System
import whiley.lang.*
import whiley.lang.Math

constant SPACE is 00100000b // ASCII code of space (' ')
constant BLACK is 01100010b // ASCII code of 'b'
constant TH is 640000 // Control the number of edges (800*800)

function wrap(int pos, int size) -> int:
  if pos>=size:
    return (size -1) - (pos - size)
  else:
    if pos <0:
      return -1 - pos
    else:
      return pos

// Perform convolution convolution on pixel at 'xCenter' and 'yCenter'
function convolution(byte[] pixels, int width, int height, int
    xCenter, int yCenter, int[] kernel) ->int:
  int sum = 0
  int kernelSize = 3
  int kernelHalf = 1
  int j = 0
  while j < kernelSize:
    int y=Math.abs((yCenter+j-kernelHalf)%height)
    int i = 0
    while i < kernelSize:
      int x=Math.abs((xCenter + i - kernelHalf)%width)
      int pixel = Byte.toInt(pixels[y*width+x])// pixels[x, y]
      // Get kernel[i, j]
      int kernelVal = kernel[j*kernelSize+i]
      //sum += pixels[x, y]*kernel[i, j]
      sum = sum + pixel * kernelVal
      i = i + 1
    j = j + 1
  return sum// 'sum' : convoluted value at pixels[xCenter, yCenter]

// Perform Sobel edge detection
function sobelEdgeDetection(byte[] pixels, int width, int height) ->
    byte[]:
  int size = width * height
  byte[] newPixels = [SPACE;size] // Output image
  // vertical and horizontal sobel filter (3x3 kernel)
  int[] v_sobel = [-1,0,1,-2,0,2,-1,0,1]
  int[] h_sobel = [1,2,1,0,0,0,-1,-2,-1]
  int x = 0
  while x<width:
    int y = 0
    while y<height:
      int pos = y*width + x
      // Get vertical gradient
      int v_g = convolution(pixels, width, height, x, y, v_sobel)
      // Get horizontal gradient
      int h_g = convolution(pixels, width, height, x, y, h_sobel)
      int t_g = v_g*v_g + h_g*h_g// Get total gradient
      if t_g > TH:
        newPixels[pos] = BLACK// Color other pixels as black
      y = y + 1
    x = x + 1
  return newPixels
```

```
61
62  // Print a pbm image
63  method print_pbm(System.Console sys, int width, int height,
64                   byte[] pixels):
65    // File type
66    sys.out.println_s("P1")
67    // Width + height
68    sys.out.print(width)
69    sys.out.print_s("␣")
70    sys.out.println(height)
71    // An array of bytes with an row of pixels in width
72    int j = 0
73    while j<height:
74      int i = 0
75      while i<width:
76        int pos = j*width + i
77        if pixels[pos] == SPACE:
78          sys.out.print(0)
79        else:
80          sys.out.print(1)
81        // Each pixel is separated by a space
82        //sys.out.print_s(" ")
83        i = i + 1
84      // Add a newline
85      sys.out.println_s("")
86      j = j + 1
87
88  // Main function
89  method main(System.Console sys):
90    // args[0]: height
91    int|null n = Int.parse(sys.args[0])
92    if n != null:
93      int width = 2000
94      int height = n
95      int size = width*height
96      // Create input pixels
97      byte[] pixels=[SPACE;size]
98      // Generate each pixels
99      int i=0
100     while i < size:
101           pixels[i]=Int.toUnsignedByte(i%256)
102           i = i + 1
103     sys.out.print_s("pixels[1000]=")
104     sys.out.println(pixels[1000])
105     byte[] newPixels = sobelEdgeDetection(pixels, width, height)
106     sys.out.println_s("Blurred␣Image␣sizes:␣␣␣")
107     sys.out.print(|newPixels|)
108     sys.out.println_s("␣bytes")
109     sys.out.print_s("newPixels[1000]=")
110     sys.out.print(newPixels[1000])
111     //print_pbm(sys, width, height, newPixels)
```

Listing B.12: LZ77 compression C program using resize array

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#define max(a, b) a ^ ((a ^ b) & -(a < b))
#define min(a, b) b ^ ((a ^ b) & -(a < b))
// Structure
typedef uint64_t nat;
typedef struct{
  int64_t len;
  int64_t offset;
} Match;
typedef uint8_t BYTE;

// Read a file from the beginning to end
BYTE* readFile(FILE *file, size_t* _size){
  // Set the file position to the beginning of the file
  rewind(file);

  // Calculate the output size
  size_t size = 0;
  while(fgetc(file) != EOF){
    //printf("%c", c);
    size = size + 1;
  }
  // Set the file position to the beginning of the file
  rewind(file);

  // Allocated byte array. Note the last char (EOF)
  BYTE* arr = (BYTE*)malloc(size*sizeof(BYTE));
  if(arr == NULL){
    fputs("fail to allocate the array at 'readAll' function in Util
    .c\n", stderr);
    exit(-2);
  }

  // Read the file to 'arr' array.
  //'fread' return the number of items read, i.e. size * sizeof(char)
  size_t result = fread(arr, sizeof(char), size, file);
  if(result != size*sizeof(char)){
    fputs("fail to read a file at 'readAll' function in Util.c\n",
    stderr);
    exit(-2);
  }

  // Update the size of 'arr' array
  *_size = size;
  return arr;
}

//
nat match(BYTE* data, size_t data_size, nat offset, nat end){
  nat pos = end;
  nat len = 0;
  while(offset < pos && pos < data_size
      && data[offset] == data[pos] && len < 255){
      offset = offset + 1;
      pos = pos + 1;
      len = len + 1;
  }
  return len;
}

```

```
62 Match findLongestMatch(BYTE* data, size_t data_size, nat pos){
63   nat bestOffset = 0;
64   nat bestLen = 0;
65   int start = max(pos - 255, 0);
66   //assert start >= 0
67   nat offset = start;
68   while (offset < pos){
69       int len = match(data, data_size, offset, pos);
70       if (len > bestLen){
71           bestOffset = pos - offset;
72           bestLen = len;
73       }
74       offset = offset + 1;
75   }
76   Match ret;
77   ret.len = bestLen;
78   ret.offset = bestOffset;
79   // Return a 'Match' object
80   return ret;
81 }
82
83 BYTE* resize(BYTE* items, size_t items_size,
84         int size, size_t* nitems_size) {
85   BYTE* nitems = (BYTE*)malloc(sizeof(BYTE)*size);
86   int i =0;
87   while(i<size){
88       nitems[i] = items[i];
89       i = i + 1;
90   }
91   *nitems_size = size;
92   return nitems;
93 }
94
95 BYTE* compress(BYTE* data, size_t data_size, size_t* _size){
96   nat pos = 0;
97   Match m;
98   size_t tmp_size=0;
99   BYTE* tmp =NULL;
100  size_t output_size=2*data_size;
101  BYTE* output = malloc(sizeof(BYTE)*output_size);
102  int size = 0;
103  while(pos < data_size){
104    m = findLongestMatch(data, data_size, pos);
105    BYTE offset = (BYTE) m.offset;
106    BYTE length = (BYTE) m.len;
107    if(offset == 0){
108      length = data[pos];
109      pos = pos + 1;
110    }else{
111      pos = pos + m.len;
112    }
113    output[size] = offset;
114    size++;
115    output[size] = length;
116    size++;
117    }
118  // Resize output array
119  tmp = resize(output, output_size, size, &tmp_size);
120  if(output!=NULL){
121    free(output);
122  }
123  output = tmp;
124  output_size = tmp_size;
125  *_size = output_size;
126  return output;
127 }
```

```
128 // Compress data
129 int main(int argc, char** args){
130   // Check if file path is passed as argument
131   if(argc != 2){
132     printf("Input␣file␣path␣is␣required\n");
133     exit(-1);
134   }
135   // Open a file
136   FILE *fp = NULL;
137   int i;
138   fp = fopen(args[1], "r");
139   size_t data_size = 0;
140   BYTE* data = readFile(fp, &data_size);
141   fclose(fp);
142   printf("Data:␣%zu␣bytes\n", data_size);
143
144   // Compress data array
145   size_t compress_data_size;
146   BYTE* compress_data = compress(data, data_size, &
        compress_data_size);
147
148   printf("Compress␣Data:␣%zu␣bytes\n", compress_data_size);
149   printf("compress_data[1000]=%d\n", compress_data[1000]);
150   free(data);
151   free(compress_data);
152   return 0;
153 }
```

Listing B.13: LZ77 decompression C Program using array list

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
// Structure
typedef uint64_t nat;
typedef uint8_t BYTE;
// Read a file from the beginning to end
BYTE* readFile(FILE *file, size_t* _size){
  // Set the file position to the beginning of the file
  rewind(file);

  // Calculate the output size
  size_t size = 0;
  while(fgetc(file) != EOF){
    size = size + 1;
  }
  // Set the file position to the beginning of the file
  rewind(file);

  // Allocated byte array. Note the last char (EOF)
  BYTE* arr = (BYTE*)malloc(size*sizeof(BYTE));
  if(arr == NULL){
    fputs("fail to allocate the array at 'readAll' function in Util
    .c\n", stderr);
    exit(-2);
  }

  // Read the file to 'arr' array.
  size_t result = fread(arr, sizeof(char), size, file);
  if(result != size*sizeof(char)){
    fputs("fail to read a file at 'readAll' function in Util.c\n",
    stderr);
    exit(-2);
  }

  *_size = size;// Update the size of 'arr' array
  return arr;
}
// If full, then double array size and store the data
BYTE* opt_append(BYTE* items, size_t items_size,
                 nat items_length, BYTE item, size_t* _size) {
  BYTE* nitems = NULL;
  size_t nitems_size=0;
  if(items_length<items_size){
    items[items_length] = item;// Update 'items' array
  }else{
    nitems_size = 2*items_size+1;
    // Create an array of 2* items array size + 1
    nitems = (BYTE*)malloc(sizeof(BYTE)*nitems_size);
    int i =0;
    while(i<items_size){
      nitems[i] = items[i];
      i = i + 1;
    }
    nitems[i] = item;
    items = nitems;
    items_size = nitems_size;
  }
  *_size = items_size;
  return items;
}

```

```
62 BYTE* resize(BYTE* items, size_t items_size,
63              int size, size_t* _size) {
64   BYTE* nitems = (BYTE*)malloc(sizeof(BYTE)*size);
65   int i =0;
66   while(i<size){
67       nitems[i] = items[i];
68       i = i + 1;
69   }
70   *_size = size;
71   return nitems;
72 }
73
74 BYTE* decompress(BYTE* data, size_t data_size, size_t* _size){
75   BYTE* items = NULL;
76   size_t items_size=0;
77   nat pos = 0;
78   nat items_length = 0;
79   BYTE* tmp = NULL;
80   size_t tmp_size = 0;
81   while ((pos+1) < data_size){
82     BYTE header = data[pos];
83     BYTE item = data[pos+1];
84     pos = pos + 2;
85     if (header == 0){
86       tmp = opt_append(items, items_size, items_length,
87                        item, &tmp_size);
88       // Free output array because it is not over-written by tmp
89       if(items != NULL && tmp != items){
90         free(items);
91         items = NULL;
92       }
93       items = tmp;
94       items_size = tmp_size;
95       items_length = items_length + 1;
96     }else{
97       int offset = (int)header;
98       int len = (int)item;
99       int start = items_length - offset;
100      int i = start;
101      while (i < (start+len)){
102        // Get byte from output array
103        item = items[i];
104        // Use array list to append item to array 'items'
105        tmp = opt_append(items, items_size,
106                         items_length, item, &tmp_size);
107        if(tmp != items && items != NULL){
108          free(items);
109          items = NULL;
110        }
111        items = tmp;
112        items_size = tmp_size;
113        items_length = items_length + 1;
114        i = i + 1;
115      }
116    }
117  }
118  //Resize the array to accurate length
119  size_t output_size = 0;
120  BYTE* output = resize(items, items_size,
121                        items_length, &output_size);
122  *_size = output_size;
123  free(items);
124  //
125  return output;
126 }
127
```

```
128 // Decompress the LZ77-compressed file
129 int main(int argc, char** args){
130   // Check if file path is passed as argument
131   if(argc != 2){
132     printf("Input file path is required\n");
133     exit(-1);
134   }
135   // Open a file
136   FILE *fp = NULL;
137   int i = 0;
138
139   fp = fopen(args[1], "r");
140   if(!fp){
141     printf("File does not exit\n");
142     exit(-1);
143   }
144   size_t data_size = 0;
145   BYTE* data = readFile(fp, &data_size);
146   fclose(fp);
147   printf("Data: %zu bytes\n", data_size);
148
149   // Decompress compressed data array
150   size_t decompress_data_size;
151   BYTE* decompress_data = decompress(data, data_size,
152                                      &decompress_data_size);
153   printf("\nDecompress Data: %zu bytes\n", decompress_data_size);
154   free(data);
155   free(decompress_data);
156   return 0;
157 }
```

Listing B.14: Sobel edge C program using `int32_t` integers

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
typedef uint8_t BYTE;
const int32_t TH = 640000;
const BYTE SPACE = 32;
const BYTE BLACK = 98;

int32_t wrap(int32_t pos, int32_t size){
    if(pos>=size){
        return (size -1) - (pos - size);
    }else{
        if (pos <0){
            return -1 - pos;
        }else{
            return pos;
        }
    }
}

int32_t convolution(BYTE* pixels, size_t pixels_size,
                    int32_t width, int32_t height,
                    int32_t xCenter, int32_t yCenter,
                    int32_t* kernel){
    int32_t sum = 0;
  int32_t kernelSize = 3;
  int32_t kernelHalf = 1;
  int32_t j = 0;
    while(j < kernelSize){
        int32_t y = abs((yCenter+j-kernelHalf)%height);
        int32_t i = 0;
        while(i < kernelSize){
          int32_t x=abs((xCenter + i - kernelHalf)%width);
          int32_t pixel = (unsigned int) pixels[y*width+x];
          int32_t kernelVal = kernel[j*kernelSize+i];
          sum = sum + pixel * kernelVal;
          i = i + 1;
        }
        j = j + 1;
    }
    return sum;
}
//Sobel edge detection
BYTE* sobelEdgeDetection(BYTE* pixels, size_t pixels_size, int32_t
    width, int32_t height, size_t newPixels_size){
    // The output image
    BYTE* newPixels = (BYTE*) malloc(sizeof(BYTE)*newPixels_size);
    int32_t i =0;
    while(i<newPixels_size){// A blank picture
        newPixels[i] = SPACE;
        i++;
    }
    // vertical and horizontal sobel filter (3x3 kernel)
    int32_t v_sobel[9]={-1,0,1,-2,0,2,-1,0,1};
    int32_t h_sobel[9]={1,2,1,0,0,0,-1,-2,-1};
    int32_t x = 0;
    while(x<width){
        int32_t y = 0;
        while(y<height){
            int32_t pos = y*width + x;
            int32_t v_g = convolution(pixels, pixels_size,
                                      width, height, x, y, v_sobel);
```

```
63              int32_t h_g = convolution(pixels, pixels_size,
64                                        width, height, x, y, h_sobel);
65              int32_t t_g = (v_g*v_g) + (h_g*h_g);
66              if(t_g > TH){// Large thresholds generate few edges
67                  newPixels[pos] = BLACK;// Color other pixels as black
68              }
69              y = y + 1;
70          }
71          x = x + 1;
72      }
73      return newPixels;
74  }
75
76  int main(int32_t argc, char** args){
77      if(argc != 2){
78          printf("Height is required");
79          exit(-1);
80      }
81      int32_t width = 2000;
82      int32_t height = atoi(args[1]);
83      printf("height=%d\n", height);
84      int32_t size = width * height;
85      printf("size=%d\n", size);
86      size_t pixels_size = size;
87      BYTE* pixels = (BYTE*)malloc(sizeof(BYTE)*pixels_size);
88      int32_t i =0;
89      while(i<pixels_size){// Initialise each pixel with SPACE
90          pixels[i] = SPACE;
91          i++;
92      }
93      i =0;
94      while(i<pixels_size){// Randomly generate each pixel
95          pixels[i] = (BYTE)(i%256);
96          i++;
97      }
98      printf("pixels[1000]=%d\n",pixels[1000]);
99      size_t newPixels_size = pixels_size;
100     BYTE* newPixels = sobelEdgeDetection(pixels, pixels_size,
        width, height, newPixels_size);
101     printf("Blurred Image sizes: %zu bytes\n", newPixels_size);
102     printf("newPixels[1000]=%d\n", newPixels[1000]);
103
104     free(pixels);
105     free(newPixels);
106     return 0;
107 }
```

B.2 LZ77 benchmark results

Table B.1: Average execution time (seconds) of LZ77 compression on medium sizes (OOM: out-of-memory, OOT: out-of-time $\geq$ *10 minutes*)

		Implementation				Speed-ups	
	Problem Size	N	N+D	C	C+D	$\frac{N}{C}$	$\frac{N+D}{C+D}$
Append array	M1x (1.58 kb)	0.085	0.030	0.013	0.013	6.5	2.3
	M5x (7.91 kb)	1.75	0.169	0.028	0.016	61.6	10.4
	M7x (11.1 kb)	3.51	0.276	0.035	0.023	99.5	12.1
	M10x (15.8 kb)	OOM	0.576	0.049	0.022	—	26.0
	M25x (39.5 kb)	OOM	6.9	0.254	0.091	—	75.8
	M50x (79.0 kb)	OOM	27.5	0.893	0.215	—	128.1
	M75x (118.6 kb)	OOM	63.0	2.0	0.485	—	130.0
	M100x (158.1 kb)	OOM	118.3	3.4	0.914	—	129.4
	M120x (189.7 kb)	OOM	175.9	31.3	1.41	—	125.2
	M125x (197.6 kb)	OOM	198.2	OOM	1.53	—	129.1
	M150x (237.2 kb)	OOM	287.9	OOM	2.31	—	124.4
	M175x (276.7 kb)	OOM	409.3	OOM	3.26	—	125.5
	M200x (316.2 kb)	OOM	548.0	OOM	4.37	—	125.4
	M225x (355.7 kb)	OOM	OOT	OOM	5.67	—	—
	M250x (395.2 kb)	OOM	OOT	OOM	7.09	—	—
	M275x (434.8 kb)	OOM	OOT	OOM	8.69	—	—
	M300x (474.3 kb)	OOM	OOT	OOM	10.4	—	—
	M325x (513.8 kb)	OOM	OOT	OOM	12.3	—	—
	M350x (553.4 kb)	OOM	OOT	OOM	14.3	—	—
	M375x (592.9 kb)	OOM	OOT	OOM	16.6	—	—
	M400x (632.4 kb)	OOM	OOT	OOM	18.9	—	—

Table B.2: Average execution time (seconds) of LZ77 compression on medium sizes (OOM: out-of-memory, OOT: out-of-time $\geq$ *10 minutes*)

		Implementation				Speed-ups	
	Problem Size	N	N+D	C	C+D	$\frac{N}{C}$	$\frac{N+D}{C+D}$
Preallocated Array	M1x (1.58 kb)	0.108	0.024	0.037	0.012	2.89	1.93
	M5x (7.91 kb)	1.71	0.18	0.016	0.013	108.4	13.9
	M7x (11.1 kb)	3.44	0.27	0.014	0.013	—	20.3
	M10x (15.8 kb)	OOM	0.52	0.015	0.016	—	33.1
	M25x (39.5 kb)	OOM	6.78	0.028	0.021	—	318.6
	M50x (79.0 kb)	OOM	26.29	0.038	0.033	—	789.9
	M75x (118.6 kb)	OOM	60.34	0.050	0.048	—	1,268.6
	M100x (158.1 kb)	OOM	117.05	0.056	0.063	—	1,852.1
	M120x (189.7 kb)	OOM	175.51	0.046	0.047	—	3,719.8
	M125x (197.6 kb)	OOM	197.22	0.068	0.069	—	2,840.7
	M150x (237.2 kb)	OOM	280.52	0.065	0.061	—	4,613.4
	M175x (276.7 kb)	OOM	395.14	0.082	0.081	—	4,908.2
	M200x (316.2 kb)	OOM	540.80	0.094	0.100	—	5,407.3
	M225x (355.7 kb)	OOM	OOT	0.110	0.098	—	—
	M250x (395.2 kb)	OOM	OOT	0.089	0.098	—	—
	M275x (434.8 kb)	OOM	OOT	0.111	0.101	—	—
	M300x (474.3 kb)	OOM	OOT	0.108	0.120	—	—
	M325x (513.8 kb)	OOM	OOT	0.138	0.118	—	—
	M350x (553.4 kb)	OOM	OOT	0.131	0.126	—	—
	M375x (592.9 kb)	OOM	OOT	0.148	0.149	—	—
	M400x (632.4 kb)	OOM	OOT	0.141	0.133	—	—

Table B.3: Average execution time (seconds) of LZ77 compression on large sizes

Problem Size	Implementation	
	C	C+D
M10000x (15.3 Mb)	2.643	2.660
M20000x (30.6 Mb)	5.538	5.319
M30000x (46.0 Mb)	8.281	7.973
M40000x (61.3 Mb)	11.043	10.616
M50000x (76.6 Mb)	13.876	13.257
M60000x (91.9 Mb)	16.705	15.944
M70000x (107.2 Mb)	19.292	18.578
M80000x (122.6 Mb)	22.065	21.241
M90000x (137.9 Mb)	24.805	23.860
M100000x (153.2 Mb)	27.637	26.518

Table B.4: Average execution time (seconds) of LZ77 decompression

		Implementation (OOM: out-of-memory)				Speed-ups	
	Problem Size	N	N+D	C	C+D	$\frac{N}{C}$	$\frac{N+D}{C+D}$
Array	M1x (1.6 kb)	0.016	0.015	0.019	0.013	0.83	1.1
	M5x (7.7 kb)	0.067	0.017	0.027	0.015	2.48	1.1
	M10x (15.3 kb)	0.152	0.037	0.060	0.020	2.52	1.9
	M25x (38.3 kb)	0.835	0.148	0.254	0.084	—	1.8
	M50x (76.6 kb)	3.237	0.592	0.927	0.222	—	2.7
	M75x (114.9 kb)	OOM	1.556	2.032	0.515	—	3.0
	M100x (153.2 kb)	OOM	3.05	3.61	0.963	—	3.2
	M125x (191.5 kb)	OOM	5.13	OOM	1.618	—	3.2
	M150x (229.8 kb)	OOM	7.76	OOM	2.450	—	3.2
	M175x (268.1 kb)	OOM	10.95	OOM	3.472	—	3.2
	M200x (306.4 kb)	OOM	14.65	OOM	4.669	—	3.1
Array List	M1x (1.6 kb)	0.011	0.015	0.013	0.013	0.84	1.2
	M5x (7.7 kb)	0.050	0.017	0.036	0.012	1.39	1.3
	M10x (15.3 kb)	0.120	0.025	0.014	0.012	8.89	2.0
	M25x (38.3 kb)	0.652	0.123	0.015	0.012	44.55	10.0
	M50x (76.6 kb)	2.485	0.393	0.013	0.014	186.22	27.3
	M75x (114.9 kb)	OOM	0.847	0.019	0.012	—	70.0
	M100x (153.2 kb)	OOM	1.627	0.013	0.015	—	108.4
	M125x (191.5 kb)	OOM	2.596	0.020	0.019	—	139.8
	M150x (229.8 kb)	OOM	3.470	0.028	0.021	—	163.4
	M175x (268.1 kb)	OOM	4.764	0.022	0.017	—	276.3
	M200x (306.4 kb)	OOM	6.611	0.047	0.013	—	511.9

Table B.5: Average execution time (seconds) of LZ77 decompression using array list on large sizes

Problem Size	C	C+D
M10000x (15.3 Mb)	0.283	0.139
M20000x (30.6 Mb)	0.534	0.274
M30000x (46.0 Mb)	0.782	0.405
M40000x (61.3 Mb)	1.031	0.534
M50000x (76.6 Mb)	1.339	0.687
M60000x (91.9 Mb)	1.568	0.797
M70000x (107.2 Mb)	1.918	0.916
M80000x (122.6 Mb)	2.167	1.044
M90000x (137.9 Mb)	2.334	1.213
M100000x (153.2 Mb)	2.594	1.332

Table B.6: Average execution time (seconds) of handwritten and generated LZ77 compression programs

Problem Size	Generated	Written	Slow-down(%)
M10000x (15.3 Mb)	2.660	2.626	1.32%
M20000x (30.6 Mb)	5.319	5.232	1.68%
M30000x (46.0 Mb)	7.973	7.834	1.77%
M40000x (61.3 Mb)	10.616	10.418	1.90%
M50000x (76.6 Mb)	13.257	12.999	1.98%
M60000x (91.9 Mb)	15.944	15.663	1.79%
M70000x (107.2 Mb)	18.578	18.271	1.68%
M80000x (122.6 Mb)	21.241	20.873	1.76%
M90000x (137.9 Mb)	23.860	23.468	1.67%
M100000x (153.2 Mb)	26.518	26.063	1.75%

Table B.7: Average execution time (seconds) of handwritten and generated LZ77 decompression programs

Problem Size	Generated	Written	Slow-down(%)
M10000x (15.3 Mb)	0.1392	0.1327	4.92%
M20000x (30.6 Mb)	0.2744	0.2658	3.22%
M30000x (46.0 Mb)	0.4047	0.3795	6.62%
M40000x (61.3 Mb)	0.5341	0.5088	4.98%
M50000x (76.6 Mb)	0.6873	0.6444	6.67%
M60000x (91.9 Mb)	0.7971	0.7572	5.27%
M70000x (107.2 Mb)	0.9157	0.8710	5.13%
M80000x (122.6 Mb)	1.0437	0.9955	4.84%
M90000x (137.9 Mb)	1.2127	1.1388	6.49%
M100000x (153.2 Mb)	1.3317	1.2507	6.48%

B.3 Sobel Edge Benchmark Results

Table B.8: Average execution time (seconds) of Sobel Edge on small sizes

Problem Size	n	Implementation				Speed-ups	
		N	N+D	C	C+D	$\frac{N}{C}$	$\frac{N+D}{C+D}$
image64x64 (4.2 kB)	1	0.026	0.020	0.011	0.015	2.38	1.36
image64x128(8.3 kB)	2	0.053	0.024	0.013	0.010	4.13	2.37
image64x192 (12.5 kB)	3	0.117	0.036	0.013	0.010	9.27	3.64
image64x256 (16.6 kB)	4	0.177	0.035	0.013	0.019	13.42	1.89
image64x320 (20.8 kB)	5	0.249	0.073	0.013	0.017	18.63	4.29
image64x384 (25.0 kB)	6	0.349	0.092	0.011	0.016	30.53	5.93
image64x448 (29.1 kB)	7	0.437	0.122	0.012	0.012	36.06	10.10
image64x512 (33.3 kB)	8	0.557	0.141	0.027	0.013	20.61	11.21
image64x576 (37.5 kB)	9	0.703	0.162	0.016	0.015	44.32	11.05
image64x640 (41.6 kB)	10	0.854	0.213	0.014	0.019	61.41	11.45

Table B.9: Average execution time (seconds) of Sobel Edge on large sizes

Problem Size	n	C	C+D	Problem Size	n	C	C+D
image2000x2000	1	0.133	0.154	image2000x22000	11	1.408	1.391
image2000x4000	2	0.268	0.263	image2000x24000	12	1.525	1.530
image2000x6000	3	0.393	0.395	image2000x26000	13	1.659	1.649
image2000x8000	4	0.526	0.523	image2000x28000	14	1.787	1.785
image2000x10000	5	0.650	0.643	image2000x30000	15	1.910	1.910
image2000x12000	6	0.775	0.778	image2000x32000	16	2.059	2.037
image2000x14000	7	0.908	0.897	image2000x34000	17	2.168	2.147
image2000x16000	8	1.027	1.019	image2000x36000	18	2.284	2.274
image2000x18000	9	1.143	1.173	image2000x38000	19	2.434	2.405
image2000x20000	10	1.271	1.270	image2000x40000	20	2.588	2.536

Table B.10: Average execution time (seconds) of written Sobel edge at `O3` optimisation

		32-bit integer (`int32_t`)				64-bit integer (`int64_t`)		
Problem Size	n	Generated	Written	Slow-down(%)	n	Generated	Written	Slow-down(%)
image2000x2000	1	0.076	0.054	41%	1	0.136	0.098	39%
image2000x4000	2	0.156	0.098	59%	2	0.273	0.172	58%
image2000x6000	3	0.231	0.134	72%	3	0.390	0.253	54%
image2000x8000	4	0.289	0.183	58%	4	0.517	0.347	49%
image2000x10000	5	0.354	0.230	54%	5	0.649	0.410	58%
image2000x12000	6	0.419	0.266	58%	6	0.761	0.494	54%
image2000x14000	7	0.499	0.311	60%	7	0.905	0.575	58%
image2000x16000	8	0.563	0.340	65%	8	1.022	0.655	56%
image2000x18000	9	0.620	0.373	66%	9	1.135	0.731	55%
image2000x20000	10	0.695	0.422	65%	10	1.258	0.804	56%
image2000x22000	11	0.765	0.461	66%	11	1.390	0.882	58%
image2000x24000	12	0.834	0.495	69%	12	1.524	0.974	56%
image2000x26000	13	0.904	0.570	59%	13	1.643	1.047	57%
image2000x28000	14	0.978	0.590	66%	14	1.771	1.125	57%
image2000x30000	15	1.065	0.627	70%	15	1.900	1.209	57%
image2000x32000	16	1.126	0.678	66%	16	2.032	1.288	58%
image2000x34000	17	1.170	0.726	61%	17	2.160	1.361	59%
image2000x36000	18	1.258	0.742	70%	18	2.279	1.441	58%
image2000x38000	19	1.333	0.790	69%	19	2.408	1.537	57%
image2000x40000	20	1.432	0.833	72%	20	2.565	1.658	55%

Appendix C

Parallel Benchmarks

C.1 Development Logs for Parallel Benchmarks

This section includes issues related to OpenMP map-reduce, Polly and Cilk parallelism.

C.1.1 OpenMP Map/Reduce

Table C.1: Complete list of Length-Offset Pairs computed by using OpenMP map/reduce program with 2 threads

POS	Length-Offset Pair
POS:0	bestLen:0 bestOffset:0
POS:1	bestLen:1 bestOffset:1
POS:2	
	ID:0 len:0 Offset:0 LocalLen[0]:0 LocalOffset[0]:0
	ID:1 len:0 Offset:1 LocalLen[1]:0 LocalOffset[1]:0
	bestLen:0 bestOffset:0
POS:3	
	ID:1 len:0 Offset:2 LocalLen[1]:0 LocalOffset[1]:0
	ID:0 len:3 Offset:0 LocalLen[0]:3 LocalOffset[0]:3

Continued on next page

Table C.1: Complete list of Length-Offset Pairs computed by using OpenMP map/reduce program with 2 threads

POS	Length-Offset Pair
	ID:0 len:1 Offset:1 LocalLen[0]:3 LocalOffset[0]:3
	bestLen:3 bestOffset:3
POS:6	
	ID:0 len:1 Offset:0 LocalLen[0]:1 LocalOffset[0]:6
	ID:0 len:1 Offset:1 LocalLen[0]:1 LocalOffset[0]:6
	ID:0 len:0 Offset:2 LocalLen[0]:1 LocalOffset[0]:6
	ID:1 len:1 Offset:3 LocalLen[1]:1 LocalOffset[1]:3
	ID:1 len:1 Offset:4 LocalLen[1]:1 LocalOffset[1]:3
	ID:1 len:0 Offset:5 LocalLen[1]:1 LocalOffset[1]:3
	bestLen:1 bestOffset:6
POS:7	
	ID:0 len:0 Offset:0 LocalLen[0]:0 LocalOffset[0]:0
	ID:0 len:0 Offset:1 LocalLen[0]:0 LocalOffset[0]:0
	ID:0 len:0 Offset:2 LocalLen[0]:0 LocalOffset[0]:0
	ID:0 len:0 Offset:3 LocalLen[0]:0 LocalOffset[0]:0
	ID:1 len:0 Offset:4 LocalLen[1]:0 LocalOffset[1]:0
	ID:1 len:0 Offset:5 LocalLen[1]:0 LocalOffset[1]:0
	ID:1 len:0 Offset:6 LocalLen[1]:0 LocalOffset[1]:0
	bestLen:0 bestOffset:0
POS:8	
	ID:0 len:0 Offset:0 LocalLen[0]:0 LocalOffset[0]:0
	ID:0 len:0 Offset:1 LocalLen[0]:0 LocalOffset[0]:0
	ID:0 len:2 Offset:2 LocalLen[0]:2 LocalOffset[0]:6
	ID:0 len:0 Offset:3 LocalLen[0]:2 LocalOffset[0]:6
	ID:1 len:0 Offset:4 LocalLen[1]:0 LocalOffset[1]:0

Continued on next page

Table C.1: Complete list of Length-Offset Pairs computed by using OpenMP map/reduce program with 2 threads

POS	Length-Offset Pair
	ID:1 len:3 Offset:5 LocalLen[1]:3 LocalOffset[1]:3
	ID:1 len:0 Offset:6 LocalLen[1]:3 LocalOffset[1]:3
	ID:1 len:0 Offset:7 LocalLen[1]:3 LocalOffset[1]:3
	bestLen:3 bestOffset:3
POS:11	
	ID:0 len:2 Offset:0 LocalLen[0]:2 LocalOffset[0]:11
	ID:0 len:1 Offset:1 LocalLen[0]:2 LocalOffset[0]:11
	ID:0 len:0 Offset:2 LocalLen[0]:2 LocalOffset[0]:11
	ID:0 len:2 Offset:3 LocalLen[0]:2 LocalOffset[0]:11
	ID:0 len:1 Offset:4 LocalLen[0]:2 LocalOffset[0]:11
	ID:0 len:0 Offset:5 LocalLen[0]:2 LocalOffset[0]:11
	ID:1 len:1 Offset:6 LocalLen[1]:1 LocalOffset[1]:5
	ID:1 len:0 Offset:7 LocalLen[1]:1 LocalOffset[1]:5
	ID:1 len:0 Offset:8 LocalLen[1]:1 LocalOffset[1]:5
	ID:1 len:1 Offset:9 LocalLen[1]:1 LocalOffset[1]:5
	ID:1 len:0 Offset:10 LocalLen[1]:1 LocalOffset[1]:5
	bestLen:2 bestOffset:11
POS:13	
	ID:0 len:1 Offset:0 LocalLen[0]:1 LocalOffset[0]:13
	ID:0 len:2 Offset:1 LocalLen[0]:2 LocalOffset[0]:12
	ID:0 len:0 Offset:2 LocalLen[0]:2 LocalOffset[0]:12
	ID:0 len:1 Offset:3 LocalLen[0]:2 LocalOffset[0]:12
	ID:0 len:2 Offset:4 LocalLen[0]:2 LocalOffset[0]:12
	ID:0 len:0 Offset:5 LocalLen[0]:2 LocalOffset[0]:12
	ID:0 len:1 Offset:6 LocalLen[0]:2 LocalOffset[0]:12

Continued on next page

Table C.1: Complete list of Length-Offset Pairs computed by using OpenMP map/reduce program with 2 threads

POS	Length-Offset Pair
	ID:1 len:0 Offset:7 LocalLen[1]:0 LocalOffset[1]:0
	ID:1 len:0 Offset:8 LocalLen[1]:0 LocalOffset[1]:0
	ID:1 len:1 Offset:9 LocalLen[1]:1 LocalOffset[1]:4
	ID:1 len:0 Offset:10 LocalLen[1]:1 LocalOffset[1]:4
	ID:1 len:1 Offset:11 LocalLen[1]:1 LocalOffset[1]:4
	ID:1 len:1 Offset:12 LocalLen[1]:1 LocalOffset[1]:4
	bestLen:2 bestOffset:12

C.1.2 Profiling Results

Table C.2: Top 5 functions of OpenMP map/reduce program

Program	Thread	%	Time (sec)	name
Sequential				
		28.14	0.09	match (LZ77:69)
		18.76	0.06	match (LZ77.c:73)
		14.07	0.05	match (LZ77.c:85)
		3.13	0.01	$findLongestMatch(LZ77.c:155)$
		3.13	0.01	$findLongestMatch(LZ77.c:164)$
OpenMP	# 1			
		21.44	0.06	$match(LZ77.c:74)$
		10.72	0.03	$match(LZ77.c:70)$
		7.15	0.02	$match(LZ77.c:27)$
		7.15	0.02	$findLongestMatch.\_omp\_fn.0(LZ77.c:240)$
		7.15	0.02	$match(LZ77.c:61)$
	# 2			
		11.12	0.03	$match(LZ77.c:27)$
		11.12	0.03	$match(LZ77.c:70)$
		7.41	0.02	$match(LZ77.c:80)$
		3.71	0.01	$findLongestMatch(LZ77.c:207)$
		3.71	0.01	$findLongestMatch.\_omp\_fn.0(LZ77.c:232)$
		3.71	0.01	$findLongestMatch.\_omp\_fn.0(LZ77.c:240)$
	# 4			
		19.06	0.04	$match(LZ77.c:27)$
		19.06	0.04	$match(LZ77.c:44)$
		9.53	0.02	$findLongestMatch.\_omp\_fn.0(LZ77.c:224)$
		9.53	0.02	$findLongestMatch.\_omp\_fn.0(LZ77.c:232)$
		9.53	0.02	$findLongestMatch.\_omp\_fn.0(LZ77.c:240)$

Our parallel OpenMP map/reduce program splits the offset iterations into a team of threads equally, so each thread has the size of offset space and spends the same amount of execution time. It seems that the OpenMP code has load-balanced schedule. The slow performance of parallel OpenMP code may result

from the overheads of creating/activated threads in OpenMP run-time.

C.1.3 Understanding LLVM Code

Polly loads Clang to compiler translates C code into LLVM code(Team, 2016) and perform the optimisation on that LLVM code. The below LLVM code snippets is parts of MatrixMult C program.

- **Module Structure** includes global variables, functions and symbol table entries (metadata).

```
; Metadata started with '!'
!1 = !DIFile(filename: "MatrixMult.c", directory: ...}
```

- **Attribute Group** specifies the module attributes referenced by all objects.

```
; define 'oninline' attribute
attributes #0 = { noinline nounwind uwtable "disable-tail-
    calls"="false" ... }
```

- **Identifiers** in LLVM has two types: local and global. Local identifiers start with '%' and global identifiers started with '@'.

```
; @R is a global 2D array of 2000 X 2000 ints
@R = common global [2000 x [2000 x i32]] zeroinitializer
; @.str is a global variable with "private" linkage.
@.str = private unnamed_addr constant [32 x i8] c"Pass␣%d␣X␣%d
    ␣matrix␣test␣case␣\0A\00"

; %conv is a local variable of 32-bit int.
%conv = trunc i64 %call to i32
```

- **Function** consists of "define" keyword.

```
; 'main' is a function with '<type> [parameter Attrs] [name]'
define i32 @main() {
; The entry point
entry:
  ; Goto 'entry.split'
  br label %entry.split

entry.split:; preds = %entry
  ;  Call 'init' function with Tail Call Optimization
  tail call void @init()
  tail call void @mat_mult()

  ; Get the address of 'A' 2D array to local register '%0'
   %0 = load i32, i32* getelementptr inbounds ([2000 x [2000 x
    i32]], [2000 x [2000 x i32]]* @A, i64 0, i64 1999, i64 1999)
```

```
15      ...
16      ; Print out 'A' array.
17      %call1 = tail call i32 (i8*, ...) @printf( ..., i32 %0, ...)
18
19    ; Return '0'
20    ret i32 0
21  }
```

- **Loop Nest** contains a loop inside another loop. The below is a loop nest with index of 'i' and 'j'. The outer loop is split into loop entry ('for.cond2.preheader'), loop exit ('for.cond12.preheader') and loop body ('for.body5'). And the loop body represents the whole inner loop, e.g. the below loop nest of 2000 iterations

```
1  for (i=0; i<2000; i++) {
2    for (j=0; j<2000; j++) {
3      A[i][j] = R[i][j];
4      B[i][j] = R[i][j];
5    }
6  }
```

can be translated to below LLVM code:

```
1  ;; Indicates the entry of outer loop
2  for.cond16.preheader:
3    ;; <result> = phi <ty> [ <val0>, <label0>], ...
4    ;; The index of the outer loop counts from 0
5    %indvars.iv5 = phi i64 [ 0, %for.cond12.preheader ], [ %
       indvars.iv.next6, %for.inc39 ]
6    br label %for.body19 ;; %indvars.iv5 := 'i'
7  ;; Represents the loop body of outer loop
8  for.body19: ; preds = %for.cond16.preheader, %for.body19
9    ;; Includes the inner loop '(j=0; j<2000; j++){Stmt(i,j)}'
10   ;; The index of inner loop counts from 0
11   %indvars.iv = phi i64 [ 0, %for.cond16.preheader ], [ %
       indvars.iv.next, %for.body19 ] ;; %indvars.iv := 'j'
12   %arrayidx23 = getelementptr inbounds [...], [...]* @R, i64 0,
        i64 %indvars.iv5, i64 %indvars.iv ;; %arrayidx23 := address of
       R[i][j]
13   %0 = load i32, i32* %arrayidx23, align 4 ;; %0 := R[i][j]
14   %arrayidx27 = getelementptr ...* @A... ;; %arrayidx27 :=
       address of A[i][j]
15   store i32 %0, i32* %arrayidx27, align 4 ;; A[i][j] = R[i][j]
16   %arrayidx31 = getelementptr ...* @R, ...;; %arrayidx31 :=
       address of R[i][j]
17   %1 = load i32, i32* %arrayidx31, align 4 ;; %1 := R[i][j]
18   %arrayidx35 = getelementptr ...* @B,...;; %arrayidx35 :=
       address of B[i][j]
19   store i32 %1, i32* %arrayidx35, align 4 ;; Write B[i][j] = R[i][j]
20   %indvars.iv.next = add nuw nsw i64 %indvars.iv, 1 ;; j := j+
       1
21   %exitcond = icmp ne i64 %indvars.iv.next, 2000 ;; Check if j !=
        2000
22   ;; if %exitcond is true, re-run 'for.body'. Otherwise, increments outer
       loop index.
23   br i1 %exitcond, label %for.body19, label %for.inc39
24 ;; The outer loop increments the index ('i').
```

```
25 for.inc39:                                        ; preds = %
       for.body19
26   %indvars.iv.next6 = add nuw nsw i64 %indvars.iv5, 1 ;; i=i+1
27   %exitcond7 = icmp ne i64 %indvars.iv.next6, 2000;; Check if 'i
       != 2000'
28   ;; If cond holds, exit the loop. Otherwise, go to the entry of outer loop.
29   br i1 %exitcond7, label %for.cond16.preheader, label %
       for.end41
30 ;; This is the loop exit and return.
31 for.end41:                                        ; preds = %
       for.inc39
32   ret void
```

- **Polly Vectorization** starts with 'polly' and the loop is transformed into vectorized loop.

- **Other LLVM Instructions** consists of *terminator instructions*, *binary instructions*, *bitwise binary instructions*, *memory instructions* and *others*.

C.2 Parallel Benchmark Results

Table C.3: Average execution time (seconds) of parallel LZ77 compression programs on 4-core (up to 8 threads) standalone machine (Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz and 16 GB memory)

			OpenMP Map/Reduce			
Problem Size	Compressed Size	**Seq**	**1 thread**	**2 thread**	**4 thread**	**8 thread**
large1x (0.57 MB)	0.15 MB	0.373	0.423	0.292	0.304	0.339
large2x (1.1 MB)	0.31 MB	0.721	0.794	0.582	0.502	0.627
large4x (2.3 MB)	0.61 MB	1.40	1.57	1.11	0.98	1.22
large8x (4.6 MB)	1.23 MB	2.79	3.10	2.23	2.02	2.37
large16x (9.2 MB)	2.45 MB	5.53	6.16	4.39	3.84	4.69
large32x (18.4 MB)	4.91 MB	11.22	12.32	8.86	7.83	9.37
large64x (36.8 MB)	9.83 MB	22.41	24.60	17.57	15.38	18.69
large128x (73.6 MB)	19.66 MB	44.08	48.99	34.58	30.36	36.41
large256x (147.2 MB)	39.35 MB	88.06	97.95	68.88	60.84	73.46
			Cilk Plus Reducer			
Problem Size	Compressed Size	**Seq**	**1 thread**	**2 thread**	**4 thread**	**8 thread**
large1x (0.57 MB)	0.15 MB	0.363	0.538	0.539	0.618	0.945
large2x (1.1 MB)	0.31 MB	0.700	1.03	0.99	1.21	1.87
large4x (2.3 MB)	0.61 MB	1.38	2.16	1.94	2.39	3.72
large8x (4.6 MB)	1.23 MB	2.70	4.10	3.84	4.80	7.39
large16x (9.2 MB)	2.45 MB	5.36	8.25	7.69	9.49	14.74
large32x (18.4 MB)	4.91 MB	10.6	16.4	15.4	19.0	29.5
large64x (36.8 MB)	9.83 MB	21.3	32.5	30.5	38.6	58.8
large128x (73.6 MB)	19.66 MB	42.6	64.9	62.3	75.0	117.5
large256x (147.2 MB)	39.35 MB	85.4	131.3	124.1	149.9	235.4

Table C.4: Average execution time (sec) of parallel LZ77 compression programs on 8-core (upto 16 threads) Google Cloud machine(Intel(R) Xeon(R) CPU@2.20GHz and 16 GB memory)

		OpenMP Map/Reduce					
Problem Size	**Seq**	**1 thread**	**2 threads**	**4 threads**	**8 threads**	**12 threads**	**16 threads**
large1x (0.57 MB)	0.45	0.50	0.49	0.63	0.88	0.65	0.64
large2x (1.1 MB)	0.90	0.99	1.01	1.22	1.74	1.06	1.22
large4x (2.3 MB)	1.79	1.99	1.89	2.49	3.42	2.10	2.40
large8x (4.6 MB)	3.59	3.95	3.88	4.91	6.73	4.37	4.75
large16x (9.2 MB)	7.24	7.95	8.78	9.86	12.98	8.36	9.20
large32x (18.4 MB)	14.69	16.33	16.64	19.86	27.61	16.89	18.59
large64x (36.8 MB)	29.32	32.29	29.78	36.83	55.38	33.74	36.72
large128x (73.6 MB)	60.82	66.46	60.50	70.39	122.52	68.31	71.73
large256x (147.2 MB)	116.57	127.99	103.72	125.20	191.09	136.20	141.33
		Cilk Plus Reducers					
Problem Size	**Seq**	**1 thread**	**2 threads**	**4 threads**	**8 threads**	**12 threads**	**16 threads**
large1x (0.57 MB)	0.44	0.68	0.81	1.16	1.84	1.55	1.62
large2x (1.1 MB)	0.90	1.37	1.60	2.77	3.62	2.99	3.19
large4x (2.3 MB)	1.74	2.70	3.84	5.47	7.15	5.98	6.40
large8x (4.6 MB)	3.48	5.42	6.71	9.76	14.33	11.99	12.83
large16x (9.2 MB)	7.22	11.22	14.46	22.90	29.37	23.96	25.71
large32x (18.4 MB)	14.26	22.28	29.11	42.60	58.03	47.86	51.30
large64x (36.8 MB)	30.52	44.93	51.26	74.10	108.94	96.19	103.40
large128x (73.6 MB)	56.19	87.99	100.86	159.68	226.84	192.92	203.90
large256x (147.2 MB)	111.52	173.82	180.62	237.65	343.36	384.74	415.28