Bound Analysis for Whiley Programs

Min-Hsien Weng[1]

Computer Science Department
Waikato University
Hamilton, New Zealand

Mark Utting[2]

Computer Science Department
Waikato University
Hamilton, New Zealand
and University of the Sunshine Coast, Australia

Bernhard Pfahringer[3]

Computer Science Department
Waikato University
Hamilton, New Zealand

Abstract

The Whiley compiler can generate naive C code, but the code is inefficient because it uses infinite integers and dynamic array sizes. Our project goal is to build up a compiler that can translate Whiley programs into efficient OpenCL code with fixed-size integer types and fixed-size arrays, for parallel execution on GPUs. This paper presents an abstract interpretation-based bound inference approach along with symbolic analysis for Whiley programs. The source Whiley program is first analyzed by using our symbolic analyzer to find the matching pattern and make any necessary program transformation. Then the bound analyzer is used to analyze the transformed program to make use of primitive integer types rather than third-party infinite integer type (e.g. using GMP arbitrary precision library). The bound analysis results provide conservative estimates of the ranges of integer variables and array sizes so that efficient code can be generated and integer overflows avoided. The bound analyzer combines the bound consistency technique along with a widening operator to give fast time of solving program constraints and of converging to the fixed point. Several example programs are used to illustrate the bound analyzer algorithm and the program transformation.

Keywords: Static Analysis, Range Analysis, Abstract Interpretation, Bound Consistency, Widening Operator, Symbolic Analysis, Pattern matching, Program Transformation.

1 Introduction

Static program analysis techniques validates the consistency between software specifications and program behaviors using mathematical methodologies. For example,

[1] Email:mw169@students.waikato.ac.nz
[2] Email: marku@waikato.ac.nz
[3] Email: bernhard@waikato.ac.nz

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

the bound consistency technique is widely used to solve the finite constraint domain problem (a.k.a constraint satisfaction problem)[17]. However, the problems of object-oriented program languages, such as side-effect problems or non-deterministic results, makes it a grand challenge[15] to create such a compiler, with automated mathematical and logical reasoning, that can verify the specifications and detect the errors at compile-time.

Whiley[20] is a new and verification-friendly programming language with the aim of resolving verification issues that arise from object-oriented programming languages. 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 instance, the use of unbounded integers causes substantial slowdown on the performance of Whiley implementations.

This paper aims to describe the design of bound analyzer along with symbolic analyzer to assist the code generator to produce behavior-predicable C code that makes use of efficient integer data types in the implementation. The main objectives are summarized as below:

- Recognize patterns of a Whiley program to make any necessary program transformation.
- Analyze the transformed program to produce the bound constraints.
- Infer the bounds using propagation rules and speed up bound analysis using the widening operator.
- Determine the efficient integer data types for the code generator.

This paper is organized as follows. Section 2 reviews some related works about static analysis, bound analysis and symbolic loop bounds. Section 3 describes the bound inference procedure, fixed-point approximation using widen operator, and pattern matching along with program transformation. Section 4 illustrates the algorithm of bound analysis with example programs and shows the performance of generated C code with/without program transformation. And the final section concludes the future work.

2 Related Work

2.1 Static Bound Analysis

Many automatic static program analyzers have been developed to improve the program correctness and produce the high-quality software, such as `ESC/Java Checker`[10] and `Microsoft Spec# Static Verifier`[2].

The static analysis using abstract interpretation, which approximates the abstract semantics of a computer program without executing all the calculation, allows the compiler to detect errors and find applicable optimization. For example, `Microsoft Research Clousot`[9] can statically check the absence of run-time errors and infer facts to discharge assertions.

However, computing the fixed-point in abstract domain is iterative and sometimes time-consuming. The abstract interpretation-based widening operator[6] can

rely on bound results at earlier iterations, and then widen the open-ended bounds to $\pm\inf$, so as to accelerate the converging time to the fixed point. But the widening operator may result in over-approximated analysis results. Thus, Su and Wanger[22] proposed the first polynomial algorithm to solve integer constraint problems without widening or narrowing operator, and showed that their method can achieve precise bounds in polynomial time whilst the termination is guaranteed.

The static analysis in `LLVM` (Low Level Virtual Machine) also gains popularity for its write-once-run-anywhere property. Campos et al.[4] used Su and Wagner's approach to implement an industrial-quality range analysis in LLVM compiler. Su's approach did not explicitly describe how to solve loop constraints, so the Campos system adapted Gawlitza approach[12] to observe the decrease or increase in cycles and then saturate the cycles using the widening operator. But compared to source-code level analyzer, the LLVM bound analyzer would more easily introduce overflow problems for the lost signed information at LLVM level. Thus, Navas, Schachte, et al.[18] designed a signedness-agnostic bound analyzer to handle both signed and unsigned LLVM variables.

Pearce[19] presented a forward propagation algorithm in Whiley to restrict the ranges of integer variables by exploiting type and loop invariant. Our approach does not require such explicit type declarations to infer the bounds, but uses the widening operator to compute the fix-point and ensure the loop termination.

2.2 Loop Bound Analysis

Loop bound analysis is a compiler optimization technique to obtain the number of loop iterations and prove the loop termination. It also can unroll the loop to reduce the compiler heap consumption. The commonly used techniques include *pattern-matching* and *counter increment*.

`CodeStatistics`, developed by Fulara et al[11], was applied with the pattern-matching technique to prove the loop termination by finding all for loop patterns in Java programs, and generating and inserting termination conditions as annotation into existing code. Their experimental 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`.

Shkaravska et al.[21] presented an counter-incremented approach 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.

However, the above approaches both fail to handle multi-path loops of different effects or non-trivial patterns. Gulwani et al.[14] used *control-flow refinement* technique to transform a multi-path loop into one or more explicit interleaving loops to simplify the analysis, and then uses *progress invariant* technique to compute precise symbolic loop bounds. Their experimental results show that Gulwani approach can find 90% of loop bounds in a large `Microsoft` product.

3 Methodology

3.1 Overview

This paper presents the bound analyzer along with symbolic analyzer to perform the analysis on Whiley programs and provide complementary analysis results to improve the quality of generated code with some optimization.

The source Whiley program is first compiled into WyIL (Whiley Intermediate Language) representation, produced by Whiley compiler, and then processed by our analyzers. Rather than from source level, the advantages of WyIL code allow the analyzers to:

- Separate from Whiley compiler and include other kinds of analysis, e.g. unique type analysis.
- Simplify the analysis for fewer control-flow constructs in WyIL.
- Optimize the program with additional program transformation and assertions without re-compilation.

Finally, the code generator translates the optimized WyIL code into bug-less and efficient C or OpenCL code that can be executed across heterogeneous platforms.

3.2 Bound Analyzer

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

3.2.1 Bound Consistency Check

Bound consistency technique[17] restricts the variables to a finite set of values and satisfies the arithmetic constraints. This technique allows the bound analyzer to propagate lower or upper bounds among variables in the form of constraints and ensure that lower bounds never exceed upper bounds.

The bound analyzer takes the function code block as input, goes through the bound constraint and inference procedure (see Algorithm 1), and produce the aggregated inferred bounds as output. It starts at the *main* function and in-lines all the function calls whenever necessary. The steps include:

CFG Construction. The analyzer processes each WyIL code semantics to build up a control flow graph (CFG) for each function and add it to a map for later use. Based on WyIL code type, the analyzer creates different types of CFG blocks (see CFG Definitions[1]), then extracts the bound constraints and place them into the CFG block.

Bound Inference. The analyzer initializes the bounds of each variable with natural number domain $[-\infty, +\infty]$ and infer the bounds from one block to another, and produces the bounds consistent with all the constraints[16]. The detailed procedure is shown in Algorithm 1.

Algorithm 1 Context-Sensitive Bound Inference Algorithm

Input: *graph*: CFG graph of function; *in_bounds*: bounds of input parameters.
Output: *bound*: bounds of function return variables.

```
1:  procedure InferBounds(graph, in_bounds)
2:      Initialize(bounds, in_bounds)
3:      iteration:=0
4:      while any bounds has changed do
5:          before_bounds := bounds
6:          for each block b do // Block order does not matter
7:              // Take the union of parents' bounds
8:              b.bounds := ⋃ { p.bounds | p ∈ b.parents}
9:              for each constraint c do // Ordered by byte-code order
10:                 if c is FunctionCall(func_name, in, out) then
11:                     bounds[out] := InferBounds(CFG(func_name), bounds[in])
12:                 else
13:                     c.inferBound(b.bounds)
14:                 end if
15:             end for
16:             // Check if lower bound ≤ upper bound.
17:             b.consistency := checkconsistency()
18:         end for
19:         iterations++
20:         if iterations % 3 == 0 then
21:             //Widen the bounds every three iterations
22:             bounds := before_bounds ∇ bounds
23:         end if
24:     end while
25:     return bounds[return_vars]
26: end procedure
```

3.2.2 Widening Operator

Abstract interpretation-based widening operator[6] is an over-approximation technique to speed up the 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 to $\pm\infty$ whilst the latter widens the bounds against a list of thresholds.

Definition 3.1 *The naive widening operator* ∇ *can be used to extrapolate the unstable bounds of an interval to* $\pm$ *infinity.*

$$\begin{aligned} \bot \nabla x &= x \\ x \nabla \bot &= x \\ [l_{\mathrm{n}}, u_{\mathrm{n}}] \nabla [l_{\mathrm{n+1}},\ u_{\mathrm{n+1}}] &= \left[l^{'}, u^{'}\right] \end{aligned}$$

where:

$$l^{'} = \begin{cases} -\infty & \text{if } l_{\mathrm{n+1}} < l_{\mathrm{n}} \\ l_{\mathrm{n}} & \text{Otherwise} \end{cases} \qquad u^{'} = \begin{cases} \infty & \text{if } u_{\mathrm{n+1}} > u_{\mathrm{n}} \\ u_{\mathrm{n}} & \text{Otherwise} \end{cases}$$

The naive widening operator observes the increases of upper bounds at each iteration and decides whether to blow out the bounds to ultimately stationary ∞. In the same manner, the operator converges decreasing lower bounds to $-\infty$. Within finite steps, the widening operator can stabilize the bounds and accelerate the bound inference time.

Table 1
Threshold Values

Threshold	**Description**	**Value**
$+\inf$		inf
$I64_{\max}$	max(long long Integers)	9,223,372,036,854,775,807
$I32_{\max}$	max(**int** Integers)	2,147,483,647
$I16_{\max}$	max(**short** Integers)	32,767
$I16_{\min}$	min(**short** Integers)	-32,768
$I32_{\min}$	min(**int** Integers)	-2,147,483,648
$I64_{\min}$	min(**long long** Integers)	-9,223,372,036,854,775,808
$-\infty$		$-\infty$

Widening with thresholds, introduced by Blanchet et al.[3], can improve the precision of interval analysis and proves the boundedness of variables. This project redefines the thresholds with integer ranges defined in `Microsoft Visual Studio C` compiler (see Table 1).

Definition 3.2 *The gradual widening operator* $\bar{\nabla}$ *goes through thresholds and finds an interval that stabilizes the bounds and reaches the fixed point.*

$$\begin{aligned} \bot \,\bar{\nabla} x &= x \\ x \bar{\nabla} \,\bot &= x \\ [l_{\mathrm{n}}, u_{\mathrm{n}}] \,\bar{\nabla}\, [l_{\mathrm{n+1}}, u_{\mathrm{n+1}}] &= \left[l^{\mathrm{th}}, u^{\mathrm{th}}\right] \end{aligned}$$

where:

$$l^{\mathrm{th}} = \begin{cases} \max(th_{\min} \in TH_{\min} \mid th_{\min} \leq l_{\mathrm{n+1}}) & if\ l_{\mathrm{n+1}} \leq l_{\mathrm{n}} \\ l_{\mathrm{n}} & Otherwise \end{cases}$$

$$u^{\mathrm{th}} = \begin{cases} \min(th_{\max} \in TH_{\max} \mid th_{\max} > u_{\mathrm{n+1}}) & if\ u_{\mathrm{n+1}} > u_{\mathrm{n}} \\ u_{\mathrm{n}} & Otherwise \end{cases}$$

The gradual widening operator broadens an increasing upper bound to the minimum of possible maximal thresholds. Otherwise, the upper bound u stays unchanged. And the operator widens the decreasing lower bound l to the maximum of minimal thresholds. The widening operator is repeatedly applied on the bounds until all the bounds have no changes. The inferred bounds allows the code generator to determine the smallest fixed-sized integer data type. For example, *short* integers can be used to store the variables in the range of $I16_{\min}$ and $I16_{\max}$.

3.3 Symbolic Analyzer

Static bound analysis provides variable bounds for the code generator to choose efficient integer data types without overflows. However, inferring a dynamic array size using a while-loop requires symbolic analysis[21]. By using pattern-matching techniques, the symbolic analyzer can infer symbolic loop bounds in Whiley program and then apply applicable transformation rules to convert one pattern into another.

3.3.1 Pattern Matching

Algorithm 2 Pattern Matching Algorithm

Input: WyIL code block
Output: Pattern

```
1: for each pattern p in available patterns do
2:     Split code block into several parts to construct the pattern p.
3:     if p is successfully constructed then
4:         return p                          // Return the matching pattern
5:     end if
6: end for
7: return NULL pattern.                      // Return no matching patterns.
```

Given a code block, the pattern matching analyzer iterates all the patterns and returns the matching one (see Algorithm 2). The symbolic analyzer has been built in with several loop patterns, including *for all* pattern, *while loop* increment or decrement pattern and *build list* pattern.

Table 2
List of Predicates of Loop Pattern

Predicates	**Return Value**
$for\_loop()$	*True* if the code block contains a *for all* loop.
$while\_loop()$	*True* if the code block contains a *while* loop.
$loop\_var(V)$	*True* if loop variable V is NOT null.
$init(V,\ Init)$	*True* if initial value of V ($Init$) is NOT null.
$cond(V,\ OP,\ B)$	*True* if while loop condition (V, OP and B) is NOT null.
$range(V,\ Init,\ B)$	*True* if the range (V, OP and B) is NOT null.
$decr(V,\ 1)$	*True* if V is only updated with a decrement of one ($V--$).
$incr(V, 1)$	*True* if V is only updated with an increment of one ($V++$).
$list\_var(L)$	*True* if the code block contains a *list* variable (L).
$list\_init(L)$	*True* if L is initialized with an empty list.
$list\_add(L,\ Item)$	*True* if L is appended with a new item ($Item$).

Each loop pattern consists of a loop variable V, initial value $Init$, loop condition (loop bound B and comparing operator OP), and the number of loop iterations $loop\_iters(V)$. The pattern predicates are defined in Table 2.

Theorem 3.3 (For All Pattern) *Let a code block contain a for all loop which iterates over a range and satisfy the below predicates:*

$$\exists v\ [\ for\_loop() \wedge range(v,\ Init,\ B)\ ] \Rightarrow loop\_iters(v) := B - Init$$

Theorem 3.4 (While Loop Decrement Pattern) *Let a code block contain a while loop with a decrementing counter and satisfy the below predicates:*

$$\exists v\ [\ while\_loop() \wedge loop\_var(v) \wedge init(v,\ Init) \wedge cond(v,\ OP,\ B) \wedge decr(v,\ 1)\ ]$$
$$\Rightarrow loop\_iters(V) := \begin{cases} Init - B & if\ OP\ is\ > \\ Init - B + 1 & if\ OP\ is\ \geq \end{cases}$$

Theorem 3.5 (While Loop Increment Pattern) *Let a code block contain a while loop with an incrementing counter and satisfy the below predicates:*

$$\exists v\ [\ while\_loop() \wedge loop\_var(v) \wedge init(v,\ Init) \wedge cond(v,\ OP,\ B) \wedge incr(v,\ 1)\ ]$$
$$\Rightarrow loop\_iters(v) := \begin{cases} B - Init & if\ OP\ is\ < \\ B - Init + 1 & if\ OP\ is\ \leq \end{cases}$$

Theorem 3.6 (Build List Pattern) *Let a code block builds up a list L and $list\_size(L)$ using a while loop and satisfy the below predicates:*

$$\begin{aligned} \exists v \exists l\ & [(while\_loop\_increment\_pattern(v) \vee while\_loop\_decrement\_pattern(v) \wedge \\ & list\_var(l) \wedge list\_init(l) \wedge list\_add(l,\ Item)] \\ \Rightarrow\ & list\_size(l) := loop\_iters(v) \end{aligned}$$

A typical *while-loop* is classified as incremented or decremented loop pattern (see Theorem 3.4 and Theorem 3.5). *Build List* pattern is also an extension of while-loop pattern with an additional list and list predicates. As the list is initialized with an empty array and inserted with only one item at each iteration, build list pattern suggests that the list size can be estimated by inferring the number of loop iterations (see Theorem 3.6).

3.3.2 Program Transformation

Program transformation technique converts one program into another valid one that makes efficient use of limited system resources, such as memory spaces. Currently, the program transformer operates on while-loop pattern and build list pattern.

Theorem 3.7 (From While-Loop Pattern to For All Pattern) *A typical while-loop increment or decrement pattern can be transformed into a for all pattern.*

$$\begin{aligned} & S_1;\quad v := init;\quad S_2;\quad \textbf{\textit{while}}(cond)\{S_3;\quad update\quad S_4;\}\quad S_5; \\ & \Rightarrow S_1;\quad S_2;\quad \textbf{\textit{for}}(v := init;\ cond;\ update)\{S_3;\quad S_4;\}\quad S_5; \end{aligned}$$

where: $S_{i=1...5}$ represents a list of statements. `init` *is the initial assignment of* `v`*,* `cond` *is the loop condition, and* `update` *is the increment or decrement of v.*

The transformer changes the while-loop structure to the for all pattern and divides the program into several parts, including pre-loop, loop header, loop body and loop exit. The pre-loop part is split into S_1 and S_2 by *init*; the loop body part is grouped into S_3 and S_4 by *update*; S_5 is the loop exit. $S_1 \dots S_5$ can be preserved and put into the transform program in order without changes (see Theorem 3.7).

Theorem 3.8 (From Build List Pattern to Build List First Pattern) *A typical build list pattern can be transformed into a static build list first pattern.*

$$\left|\begin{array}{l} S_1;\quad v := |ls|;\quad r := [];\quad S_2; \\ \textbf{\textit{while}}(v > 0)\ \textbf{\textit{where}}(v \le |ls|)\{ \\ \quad S_3;\quad v++;\quad r += [ls[v]];\quad S_4; \\ \}\quad S_5; \end{array}\right| \Rightarrow \left|\begin{array}{l} S_1;\quad v := |ls|;\quad r_c := |ls|;\quad r := ls;\quad r_s := 0;\quad S_2; \\ \textbf{\textit{while}}(v > 0)\ \textbf{\textit{where}}((v \le |ls|)\&\&(r_s \ge 0))\{ \\ \quad S_3;\quad v++;\quad r[r_s] = ls[v];\quad r_s++;\quad S_4; \\ \}\quad assert\ r_s == r_c;\quad S_5; \end{array}\right|$$

where: `v` *is the loop variable.* `ls` *is the input list.* `|ls|` *is the length of ls.* `r` *is the output list.* r_c *is the capacity of r.* r_s *is the size of r.* $S_{i=1\dots5}$ *is the list of statements.*

A build list pattern can be transformed to the *build list first* pattern for better performance. Rather than starting with an empty array, the new build list pattern first copies the input list and then gradually fills each item in the list using the loop. Due to the use of fixed-sized list, the transformed program avoids re-sizing list capacity and gains speedups from in-place update.

4 Evaluation

The bound analyzer algorithm is illustrated with three test cases, including `nested if-else`, `while-loop` and `loop-bounded list` programs. Each contains a 'main' method along with one or two functions.

An external industrial analyzer, **Frama-C** value analyzer[7], is used to verify our bound analyzer and remove the chances of false alerts from our analyzer because it has been developed and used for more than 10 years and is relatively bug-free. To make use of Frama-C analyzer, the Whiley program is first translated into the provisional C code, using the *long long* type for every integer without type checking, and then perform the range analysis.

4.1 Test Case: Nested If-Else Program

```
1: function f(int x) → int:
2:    if x < 10:
3:       return 1
4:    else:
5:       if x > 10:
6:          return 2
7:    return 0
8: method main(System.Console sys) → void:
9:    sys.out.println(f(10))
```

Table 3
Bound Results

Domain	Naive	Gradual	Frama-C
D(f(10))	0	0	0
D(f(11))	2	2	2
D(f(1212))	2	2	2
D(f(-1212))	1	1	1

This test case shows that the bound analyzer produces context-sensitive bounds. For example, when the bound analyzer encounters a function call, it passes the bounds of x ($[10 \ldots 10]$) to 'f' function and then performs the bound inference procedure (see Algorithm 1) to obtain the bounds of return value of 'f' function and propagate the return bounds back to 'main' function. The results in Table 3 show the bound analyzer can provide exactly the same results as Frama-C.

4.2 Test Case: While Loop Program

// Sum up all the integers from 0 upto the given limit.

```
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
9: method main(System.Console sys)
   → void:
10:   sys.out.println(f(50000))
```

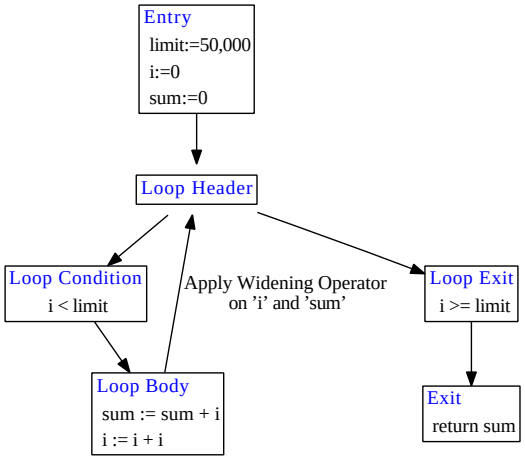

Fig. 1. Control Flow Graph

Table 4
Bound Results of While Loop Program

Domain	Naive Widening Operator	Gradual Widening Operator	Frama-C	Frama-C(slevel= 50,000)
$D(limit)_{\text{entry}}$	[50,000...50,000]	[50,000...50,000]	[50,000...50,000]	[50,000...50,000]
$D(i)_{\text{loop\_header}}$	[0... 50,000]	[0... 50,000]	[0... 50,000]	[0...50,000]
$D(i)_{\text{exit}}$	[50,000...50,000]	[50,000...50,000]	[0...50,000]	[0...50,000]
$D(sum)_{\text{exit}}$	[0... inf]	[0... inf]	[− inf ... inf]	1,249,975,000

This test case shows the naive and gradual operator can widen the bounds and force the bound analyzer to terminate the while-loop in f function. Table 4 shows that on this simple example the naive and gradual widening operators give the same results, but on some more complex loop updates, such as $x = x/2 + 100$[3], the gradual widening operator gives more accurate results than the naive widening operator.

Compared to our bound analyzer, Frama-C provides a more precise range over 'i' variable and with 'slevel' option set to 50,000 [4] produces concrete results.

[4] The *slevel* option allows FramaC to unroll the loop[5], and thus produce accurate results. But it has the side effect of long computing time and does not help on many programs because it is performing concrete execution rather than abstract interpretation.

4.3 Test Case: Loop-bound List Program

Original Program

```
1: function reverse([int] ls) → [int]:
2:    int i = |ls|
3:    [int] r = []
4:    while i > 0 where i ≤ |ls|:
5:       i = i - 1
6:       r = r ++ [ls[i]]
7:    return r
8: method main(System.Console sys)
   → void:
9:    [int] xs = []
10:   int limit = 300000000
11:   //Create input list using a for loop
12:   for i in 0.. limit:
13:      xs = xs ++ [i]
14:   [int] rs = reverse(xs)
15:   sys.out.println(rs)
```

Transformed Program

```
1: function reverse([int] ls) → [int]:
2:    int i = |ls|
3:    int r_capacity = |ls|
4:    [int] r = ls
5:    int r_size = 0
6:    while i > 0 where i ≤ |ls|
      && r_size >= 0:
7:       i = i - 1
8:       r[r_size] = ls[i]
9:       r_size = r_size +1
10:   assert r_size == r_capacity
11:   return r
12:..
13;//main() remains the same.
```

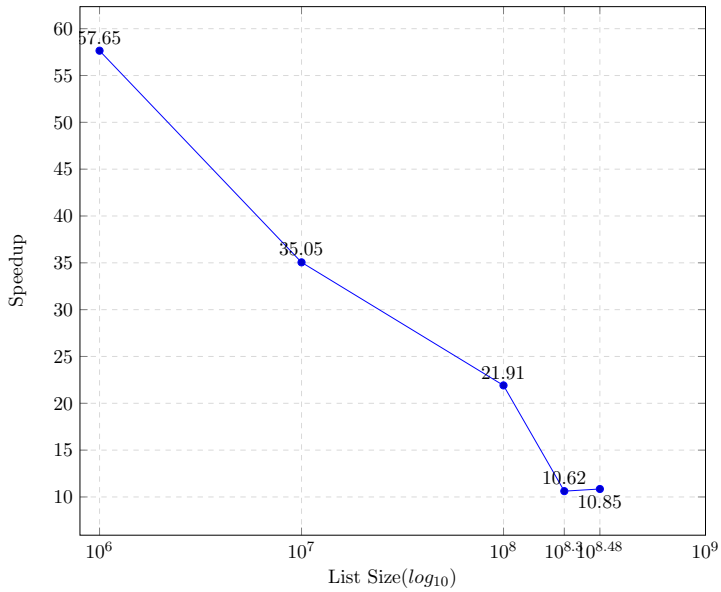

Fig. 2. Performance of Program Transformation

The original 'reverse' function is a typical build list pattern. Inside the loop, each single insertion adds one item to the list and thus constraints the list size to the loop bound (see Theorem 3.6). The symbolic analyzer first performs the pattern matching. Once the program is matched with build list pattern, it transforms the original program and generates a new program (see Theorem 3.8). The transformed program initializes the list with the input list, instead of an empty list, and imposes bounds on the list size with additional *r_capacity* and *r_size* variables and an assertion so that out-of-bound errors can be avoided.

The code generator translates the program into C code using *long long* integers rather than arbitrary precision GNU MP[13] integers because the primitive integers would give more speedups to the generated implementation. Future versions of the code generator will generate short or int types where the inferred bounds of a variable fit within that range, and will reject Whiley programs that contain variables whose bounds are larger than long long.

The benchmark experiment repeatedly runs the generated C code of original and transformed 'reverse' function for 10 times against 5 kinds of list sizes: 1 million, 10 million, 100 million, 200 million and 300 million. The average execution time ignores the first run and averages the remaining ones. The speedup is the average execution time of the original reverse function, divided by the average execution time of the transformed reverse function.

Benchmarks are run on the i5 2.4GHz CPU machine with 8G memory and the speedup performance is plotted in Figure 2. The experiment results show that the program transformation improves the efficiency of the program upto 57 times faster but the speedup reduces to 10 times as the list size increases to 200 million.

5 Conclusions

The bound analyzer and symbolic analyzer analyze a Whiley program at byte-code level and provides complementary analysis results for the code generator to produce efficient and safe C code.

In *Nested If-else* test cases, the bound results show that the bound analyzer propagates the bounds for each function call and produces context-sensitive bounds that are as precise as the bounds from the industrial Frama-C. In the *while-loop* test case, the bounds analysis result show that the bound analyzer using the gradual widening strategy achieves more precise bound results than Frama-C (without unroll-loop option). In the *Dynamic-sized List* test case, the performance chart shows that the program transformation speeds up the generated C code, but allows the bound analyzer to replace the dynamic list by a fixed-size array with in-place updates. This will be a good basis for generating parallel code in the future.

The symbolic and bound analysis enable the code generator to make use of fixed-sized integers and fixed-sized lists in translation so the generated code avoids integer overflows and out-of-range list errors and reduces memory usage. This makes our project similar to the w constraint operator[8]. But our gradual widening operator increases the precision of bound analysis, and the support for program transformations improves the efficiency of generated code.

The future work includes solving the extra copying of data types and the use of inefficient data structures. And the code generator needs further improvement and code optimization to generate the OpenCL code for parallel execution on GPUs.

Acknowledgement

Thanks for Dr. David J. Pearce's technical support and to Google for some funding support for this project via a grant to Dr Utting.

References

[1] 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.

[2] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# Programming System: An Overview. In *CASSIS 2004, Construction and Analysis of Safe, Secure and Interoperable Smart devices*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69. Springer, 2005.

[3] 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.

[4] 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 LuisSoares Barbosa, editors, *Programming Languages*, volume 7554 of *Lecture Notes in Computer Science*, pages 42–56. Springer Berlin Heidelberg, 2012.

[5] Loc Correnson, Pascal Cuo q, Florent Kirchner, Virgile Prevosto, Armand Puccetti, Julien Signoles, and Boris Yakob owski. Frama-C User Manual. User manual, Software Safety Laboratory, CEA LIST, March 2014.

[6] Agostino Cortesi and Matteo Zanioli. Widening and Narrowing Operators for Abstract Interpretation. *Computer Languages, Systems & Structures*, 37(1):24–42, 2011.

[7] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C: A Software Analysis Perspective. In *Proceedings of the 10th International Conference on Software Engineering and Formal Methods*, SEFM'12, pages 233–247, Berlin, Heidelberg, 2012. Springer-Verlag.

[8] Tristan Denmat, Arnaud Gotlieb, and Mireille Ducass. An Abstract Interpretation Based Combinator for Modelling While Loops in Constraint Programming. In Christian Bessire, editor, *Principles and Practice of Constraint Programming CP 2007*, volume 4741 of *Lecture Notes in Computer Science*, pages 241–255. Springer Berlin Heidelberg, 2007.

[9] Manuel Fahndrich and Francesco Logozzo. Static Contract Checking with Abstract Interpretation. In *Proceedings of the Conference on Formal Verification of Object-oriented Software (FoVeOOS 2010)*. Springer Verlag, October 2010.

[10] 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.

[11] 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.

[12] 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.

[13] Torbjrn Granlund and the GMP development team. GNU MP — The GNU Multiple Precision Arithmetic Library. Available: https://gmplib.org/gmp-man-6.0.0a.pdf. Accessed: 25 March 2014.

[14] Sumit Gulwani, Sagar Jain, and Eric Koskinen. Control-flow Refinement and Progress Invariants for Bound Analysis. *SIGPLAN Not.*, 44(6):375–385, June 2009.

[15] Tony Hoare. The Verifying Compiler: A Grand Challenge for Computing Research. *J. ACM*, 50(1):63–69, January 2003.

[16] 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.

[17] K. Marriott and P. J. Stuckey. *Programming with Constraints: An Introduction*. Adaptive Computation and Machine. MIT Press, 1998.

[18] 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.

[19] David J. Pearce. Integer range analysis for whiley on embedded systems. In *the IEEE/IFIP Workshop on Software Technologies for Future Embedded and Ubiquitous Systems*, 2015.

[20] David J. Pearce and Lindsay Groves. Whiley: A platform for research in software verification. In Martin Erwig, RichardF. Paige, and Eric Van Wyk, editors, *Software Language Engineering*, volume 8225 of *Lecture Notes in Computer Science*, pages 238–248. Springer International Publishing, 2013.

[21] 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.

[22] 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.