Xev-GMP: Automatic code generation for GMP multiple-precision code from C code

Toshiaki Hishinuma *, Takuma Sakakibara†, Akihiro Fujii‡, Teruo Tanaka§ and Shoichi Hirasawa¶
*University of Tsukuba, Ibaraki, Japan, Email: hishinuma@slis.tsukuba.ac.jp
†Kogakuin University, Tokyo, Japan
‡Kogakuin University, Tokyo, Japan, Email: fujii@kogakuin.ac.jp
§Kogakuin University, Tokyo, Japan, Email: teru@kogakuin.ac.jp
¶Tohoku University, Miyagi, Japan, Email: hirasawa@sc.cc.tohoku.ac.jp

Abstract—**We propose directive-based automatic code generation for a multiple-precision code from a C code with double precision. The multiple-precision code uses the GNU Multiple Precision Arithmetic Library (GMP). Our code generation functions can be separated into binary operations by automatically creating temporary variables, transforming C mathematical functions into corresponding GMP functions, and managing functions that return a double-precision value. Our proposed system enables users to check the accuracy dependency of many algorithms by adding a few directives to C codes with double precision.**

1. Introduction

In many cases, the kernel of a numerical simulation is the solution of large and sparse systems of linear equations. Krylov subspace methods are well-known algorithms for this solution, but these methods diverge, stagnate, and increase iterations because of rounding errors. High-precision arithmetic may improve the convergence of these methods[1]; however, it is very costly. Computational costs have been mitigated by advances in computing hardware; however, programming costs remain a problem.

The GNU Multiple Precision Arithmetic Library (GMP)[2] is a popular multiple-precision arithmetic library in C. GMP does not require any special hardware and runs on general-purpose processors. However, programming using GMP presents several serious problems. The functions in GMP cannot return multiple-precision data types. Operations need to be separated into binary operations using temporary variables. The programming cost of a C code using GMP (a GMP code) is more than that of a C code written in double precision (a C code). The "multiple-precision arithmetic" of GMP does not include double-precision arithmetic. Using GMP, one cannot employ a combination of double and multiple precisions without rewriting the C code. Moreover, transposing a C code into a GMP code is an expensive process.

Bailey compiled a basic multiple-precision operation and the transcendental library MPFUN[3]. The Omni OpenMP Fortran 77 Compiler[4] can declare multiple-precision variables on the basis of GMP. The MPFR Library[5] can use the operator overloading technique in C++. Programs using the abovementioned libraries and compiler are not compatible with all C programs, and it is necessary to rewrite the source code.

By generating a source code using a directive, multiple functions can be obtained via the management of a single source code. This model's advantage is its interoperability. Here, we propose directive-based automatic code generation for a multiple-precision code using GMP that is generated from a C code written in double precision. These functions will enable users to maintain a single source code for double- and multiple-precision arithmetics.

We developed "Xev-GMP" directive-based code generation functions that can automatically create temporary variables, transform C mathematical functions into corresponding GMP functions, and manage functions that generate the call-by-pointer method returning a double-precision value.

2. Design concept of Xev-GMP

2.1. GNU Multiple Precision Arithmetic Library

The differences between the GMP code and the C code are as follows.

- All operations in GMP are implemented by a function call.
- A function cannot return a multiple-precision value in GMP. It must declare a "void" type.

GMP programming requires dividing an expression composed of multiple terms into binary operations.

The multiple-precision floating-point GMP data-type is defined as "mpf_t," which consists of a 1-bit sign part, an 11-bit exponent part, and an arbitrary bit signification part. On a 64-bit system, the significand part of "mpf_t" is implemented using a 64-bit integer array. The significand part of GMP is at least 64-bit.

2.2. Xevolver: an XML-based code translation framework

Recently, Takizawa et al. developed the Xevolver, which is an XML-based code translation framework. Figure 1

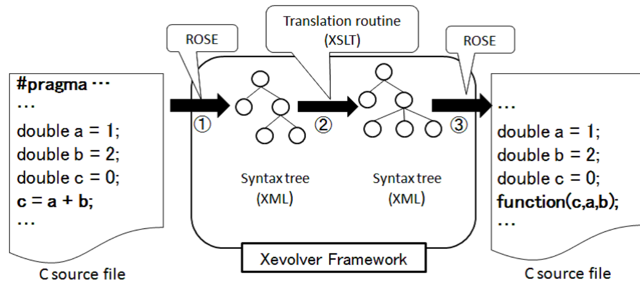

Figure 1. Overview of the code translation using Xevolver.

shows an overview of the code translation using Xevolver. This framework can make user-defined compiler directives using XSL transformations (XSLT)[7]. Using Xevolver, the user can easily create code generation functions. Code generation with Xevolver is performed with the following three steps.

(1) A source code is parsed, and a parse tree is output as an XML document[8] using the ROSE compiler infrastructure (ROSE)[9].
(2) Code is generated using the code generation functions of the directives in XSLT from an XML document, and the generated parse tree is output as an XML document.
(3) The Xevolver unparses the parse tree to generate a modified version of the C code using ROSE.

"Xev-GMP" is a directive-based automatic code generation method for the GMP code that is generated from the C code. A user can generate a GMP code from a C code by writing "Xev-GMP" directives. To set the accuracy of each value, we prepared two directives:

(A) #pragma xev gmp default(prec)
The user writes this directive in the header part of a C source file. This specifies the default precision of each value. When the user writes this directive, all processes related to "mpf_t" are generated.
(B) #pragma xev gmp set(prec)
The user writes this directive around the declaration part of the variable. This specifies the arbitrary precision of the variables. In the present implementation, this is written in only one place in a C source file.

"Prec" is an integer variable.

3. Implementation of Xev-GMP

Table 1 shows a list of generation functions in "Xev-GMP." We developed translation functions including the basic functions in C. In the next section, we explain the typical code generation functions.

3.1. Variable declaration

Programming using GMP requires calling the functions for initialization, declaration, and releasing of the data type of "mpf_t." Four functions, "mpf_set_default_prec(precision)," "mpf_init(dest)," "mpf_set_d(dest, value)," and "mpf_clear(dest)," are used. "Dest" is the "mpf_t" type destination value of a function, "precision" is the integer value of the bit size of a significand part, and "value" is the double-precision value.
We show these details as follows.

- "mpf_set_default_prec(dest)" sets the default precision. All subsequent calls to "mpf_init" use this precision.
- "mpf_init(dest)" initializes the value to zero in the default precision.
- "mpf_set_d(dest, value)" assigns a new value to the already initialized double-precision "value."
- "mpf_clear (dest)" releases the space occupied by the value.

We developed these generation functions for variable declarations with the following five steps.

(1) Change the declaration of variables in double precision to a data type of "mpf_t."
(2) Insert the initialization function "mpf_init" after the declaration part of the variables.
(3) If "#pragma xev gmp set" has been used, insert the initialization function "mpf_init2" after the declaration part of the variable instead of "mpf_init."
(4) If the initialization has been performed at the time of the declaration of the double-precision value, insert the assignment function "mpf_set_d" after the initialization.
(5) Insert the releasing memory function "mpf_clear" before "return."

TABLE 1. LIST OF THE GENERATION FUNCTIONS IN XEV-GMP.

#	Generation functions	Remarks
1	Including "gmp.h"	
2	Specified precision	
3	Mixed precision	Only at one place in a file.
4	Variable declaration	
5	Variable initialization	
6	Array	
7	Comparison operator	A ternary operator is impossible.
8	Arithmetics	"+," "-," "*," and "/."
9	Standard I/O	
10	File I/O	
11	Record	A nested record is impossible.
12	Mathematical functions	e.g., log(), sqrt()
13	User defined functions	
14	Other functions	e.g., atof(), omp_get_wtime()
15	Arithmetic in function operand	e.g., log(a+b)

3.2. Arithmetic operations

A function in GMP is "mpf_add(dest, src1, src2)," and that of multiplication is mpf_mul(dest, src1, src2). This means that dest = src1 + / * src2. "Dest," "src1," and "src2" are "mpf_t" type variables.
The left part of Figure 2 shows a syntax tree of the expression "a = b + c * d" and that the expression contains

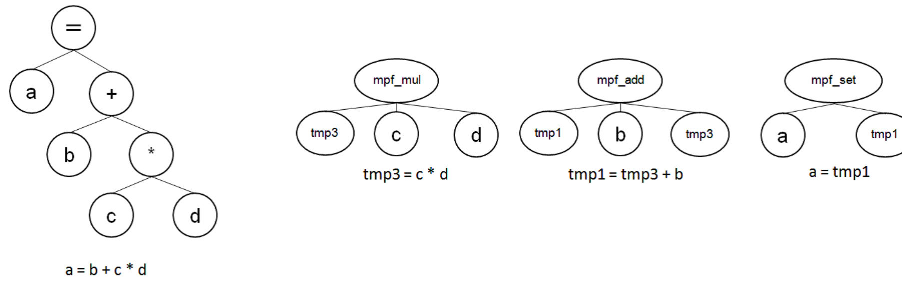

Figure 2. Syntax tree of “a = b + c * d” (left: C code, right: GMP code).

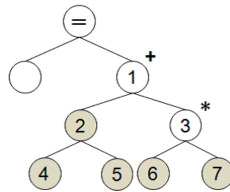

Figure 3. Numbering rules for temporary variables.

three operators, “+,” “*,” and “=.” In the GMP code, the process must be divided into two binary arithmetic operations: “mpf_mul” and “mpf_add” for a = b + c * d. Our code generation function assigns each operation to a corresponding function, as in the right side of Figure 2. Therefore, the following GMP code is generated.

```
mpf_mul(tmp3, c, d);
mpf_add(tmp1, b, tmp3);
mpf_set(a, tmp1);
```

“Mpf_set(dest, src1)” assigns a new value to the already initialized “mpf_t” type variable.

There are two problems for a translation generating GMP arithmetic operations from a C code. The first is the numbering of the temporary variables, and the second is the declaration and releasing of the variables.

To solve these problems, we perform the conversion in two steps. In the first step, we recursively search a syntax tree and assign temporary variables according to certain rules. In the second step, we search the C code for temporary variables. Then, we add the declaration and releasing operations of these variables.

We use the position number of the depth and width in the arithmetic tree to count the temporary variables. We use the number corresponding to the position in the complete binary tree so that we can reuse the temporary variables in different arithmetic trees.

First, we number all the nodes of the arithmetic tree and then assign a number to the operators. This approach can reuse temporary variables in different lines and reduce the number of necessary temporary variables. However, it calls extra “mpf_set” functions for the “=” operation.

Figure 3 shows the numbering rules of the temporary variables. The “=” operation is the root node. The left-child node of the root node is the only destination value. We define the right-child node of the root node as number 1. Then, we form a temporary complete binary tree and number the nodes such that the root node is number 1. Finally, we delete the unused nodes (the gray nodes in Figure 3).

For example, we create two temporary variables to represent the operators in the expression “b * c + d.” The operators “+” and “*” correspond to nodes 1 and 3 in the syntax tree in Figure 3. Therefore, the temporary variables “tmp1” and “tmp3” are added.

Next, we apply the second step. We append the declaration and initialization of the temporary variables in the header part of the function and release the variables before the “return.”

3.3. Function calls

There are three types of functions:

(a) Functions that have been implemented by GMP, e.g., “mpf_sqrt()” and “mpf_fabs().”
(b) User-defined functions; and
(c) Other functions, e.g., atof() and omp_get_wtime().

We made a function list for (a). For example, we generate “mpf_sqrt(a, b)” from “a = sqrt(b)” using this list.

If (b) returns a double-precision value, it is necessary to change that value into an argument of the function. We implement the generation function of (b) with the following three steps.

step 1 Add the return value in the argument of a user-defined function.
step 2 Before the “return” sequence of the function, store the value to the returned argument variable.
step 3 Change the argument of the “return” sequence to zero.

As a result, “x = function(*x, a)” can be generated from “function(a).”

(c) returns a double-precision value, and it cannot be applied in GMP. For example, GMP does not have a time-measuring function. One time-measuring function is OpenMP’s “omp_get_wtime()”, which returns a double-precision value. In this case, we directly use these functions

with “mpf_set_d.” “Mpf_set_d(time, omp_get_wtime())” is generated from “time = omp_get_wtime().”

Using these generation functions, the following code generation is performed.

(1) Perform the “mpf_sqrt” function using (a).
(2) Create the declarations and release the temporary variables.
(3) Change the format for calling the functions, which is generated by (b).
(4) Set the return value to the argument of the user-defined function that was generated by (b).
(5) Perform the “mpf_set_d” function for “omp_get_wtime” using (c).

3.4. Evaluation of Xev-GMP using the SOR solver

We evaluated the implementation cost and the result of the GMP code generated using “Xev-GMP.” The Xevolver framework was XevXML commit ID: [6354fb6] (20150618). “Xev-GMP” ver. 0.9-beta, ROSE ver. 3-0.95a-20584, and gcc-4.4.7 were used. The compiler options -O3 and -lgmp were used to link the gmp-6.0.0 library on the CentOS 6.2.

We used the SOR solver program included in the ANSI C version of SciMark 2.0[10].

We made two changes in the C code. First, we added the ”#pragma xev gmp default(128)” in the header part of the C code. Second, we changed the type of the function to double precision from “void,” which returned “omega_over_four.”

As a result, “Xev-GMP” generated 65 lines of GMP code from 32 lines of C code by inserting only a single directive line.

We examined the execution result of the GMP code by comparing the results of the C code and the 128-bit GMP code. The differences in the corresponding values in the GMP and C codes were less than 1.0E-8. Therefore, we concluded that the generated GMP code ran correctly.

The code generated with “Xev-GMP” used six temporary variables and four “mpf_set.” If the code were optimal, it would require two temporary variables and would not need “mpf_set.”

3.5. Discussion

Xev-GMP requires only the information of the precision in a single directive line. The programming cost of using Xev-GMP is small. It does not require knowledge of GMP. A user can use high-precision arithmetic without being conscious of GMP programming and does not need to have knowledge of GMP programming.

Our functions can reuse temporary variables in different lines and reduce the number of necessary temporary variables. However, it calls extra “mpf_set” functions for the “=” operation. Therefore, it is not an optimal code. The elapsed time of the code generated by Xev-GMP is 2% slower than the optimal code, and it requires four extra GMP variables. We believe that the influence of this problem on the performance and extra memory data space required is small.

Xev-GMP enables users to check the accuracy dependency of many algorithms by adding a few directives to C codes with double precision.

4. Conclusions

We proposed directive-based automatic code generation for a GMP multiple-precision code from a C code with double precision.

Our code generation functions recursively search a parse tree and divide the process into two steps. They automatically create temporary variables, transform C mathematical functions into corresponding GMP functions, and manage functions that return a double-precision value.

With our proposed system, it is possible to generate a GMP code from an ordinary C code by adding only a single directive line. This directive can reduce the programming cost. This code generation enables users to evaluate the accuracy-dependent behavior of their codes without rewriting them. It is also possible to generate a GMP code from a C code by specifying only the required precision as the directive.

In the future, we will implement “Xev-GMP” searching functions to search across multiple C source files with a more flexible precision input user interface.

Acknowledgments

This work was partially supported by JST CREST “An Evolutionary Approach to Construction of a Software Development Environment for Massively-Parallel Heterogeneous Systems,” JSPS KAKENHI Grant Number 25280041, and JSPS KAKENHI Grant Number 25330144.

References

[1] Tomonori Kouya, A Highly Efficient Implementation of Multiple Precision Sparse Matrix-Vector Multiplication and Its Application to Product-type Krylov Subspace Methods, International Journal of Numerical Methods and Applications, Vol. 7, Issue 2, pp. 107-119, 2012.

[2] The GNU Multiple Precision Arithmetic Library, https://gmplib.org/.

[3] MPFUN, http://www.davidhbailey.com/dhbsoftware/.

[4] Omni OpenMP Fortran 77 Compiler, http://www.hpcs.cs.tsukuba.ac.jp/omni-compiler/old-doc/omf77.html/.

[5] Fousse Laurent et al., MPFR: A Multiple-precision Binary Floating-point Library with Correct Rounding, ACM Trans. Math. Softw., Volume 33, Issue 2, Article No. 13, pp. 1-14, 2007.

[6] Hiroyuki Takizawa, Shoichi Hirasawa, et al., Xevolver: An XML-based Code Translation Framework for Supporting HPC Application Migration, IEEE International Conference on High Performance Computing, pp. 1-11, 2014.

[7] XSL Transformations, http://www.w3.org/TR/xslt/.

[8] Extensible Markup Language, http://www.w3.org/XML/.

[9] ROSE compiler infrastructure, http://rosecompiler.org/.

[10] SciMark 2.0, http://math.nist.gov/scimark2/.