Chapter 6. Memoization Approach

Loïc Besnard

Univ. Rennes, CNRS, IRISA, France
loic.besnard@irisa.fr

Abstract

This chapter presents a technique, called memoization, that catches results of pure functions and retrieves them instead of recomputing a result to optimize applications for energy efficiency. The definition of LARA aspects allows to the user to apply the memoization in a very easy way to C and C++ applications.

1 Introduction

Optimizing applications for energy efficiency is a challenge of the ANTAREX project. We introduce in this chapter a memoization technique that saves the results of computations so that future executions can be omitted when the same inputs repeat.

The remainder of this chapter is organized as follows. We describe first the main goal of the library developed and its main features. Then, the LARA aspects developed in ANTAREX that ensure the interface between the user and the library are presented. In a third section, experimental results are presented.

2 Principles of the memoization

Performance can be improved by caching results of pure functions (i.e. deterministic functions without side effects), and retrieving them instead of recomputing a result [1] [2]. This technique may be applied to C functions and C++ memoizable methods. It takes into account the mangling[3], the overloading, and the references to the objects. Consider a memoizable C function `foo` as shown in Figure 1. The memoization consists in:

- the insertion of a wrapper function `foo_wrapper` and an associated table. The elements of the internal tables are indexed with a hash calculated from the call arguments of the memoized function.
- The substitution of the references to foo by `foo_wrapper` in the application.

```c
float foo (float p) {
  /* code of foo without side effects */
}

float foo_wrapper(float p)  {
  float r;
  /* already in the table ? */
  if (lookup_table(p, &r)) return r;
      /* calling the original function */
      r = foo(p);
  /* updating the table or not */
  update_table(p, r);
  return r;
}
```

Figure 1: A memoizable C function and its wrapper.

To be memoized, the interface of a function (or a method) must verify the following properties:

- the function/method has at most 4 arguments of same type T,
- the function/method returns a data of type T,
- T is an element of 'double', 'float', 'int' types.

The memoization operation of a function/method is parametrized by the following informations:

- The size of the internal table.
- The initialization of the internal table by the content of a file.
- The saving or not of the results. At the end of the execution, the data of the table is saved in a file. These results may be used as input for an next execution.
- The replacement policy to be used in case of index conflict. The user must specify if the value of the table must be or not replaced. It may also specify a 'full off line' policy when an initial table is provided: in this case the table is never updated.
- The approximation parameter that allows to not distinguish very near parameter values.

Based on the selected memoizable functions, the framework generates (see Figure 2) a new version of the application enhanced with memoization support by relying on the Clava source-to-source compiler. The framework is also responsible to generate the C library that contains the core of the memoization implementation, which is linked with the newly generated application. The LARA aspects generate required informations in a file. For each function/method to memoize, the file contains a definition such that

DEF(CL, F, Fwrapper, N, Type, Approx, InFile, FullOffLine, OutFile, Replace, Tsize)

that define the associated parameters:

- CL: a codification 0 (math function), 1 (C++), 2(C).
- F: the name of the function/method to memoize with mangling.
- Fwrapper: associated wrapper function with mangling.

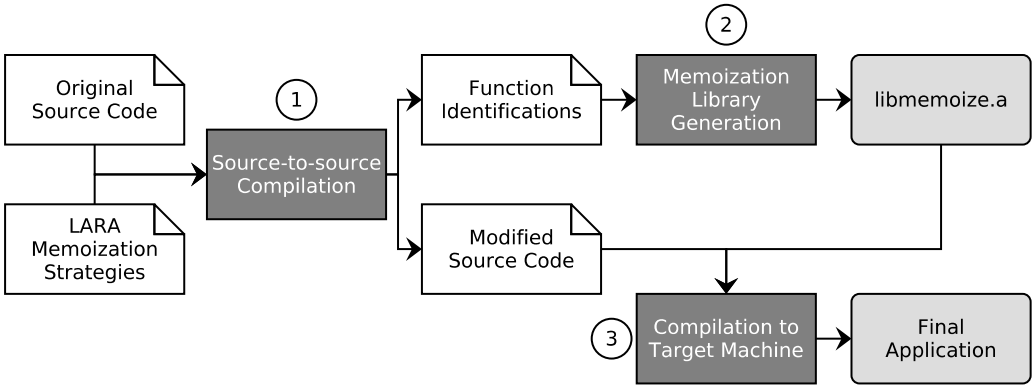

Figure 2: Tool flow of the memoization framework.

- N: number of arguments of type Type
- Type: the type restricted to `int, float ,double`.
- Approx: number of bits to remove for approximation, restrited to `float` and `double` types.
- InFile: name of a file to load or none. When it is specified, the table is initialized with the content of the file using the "hexa floating point" format.
- FullOffLine: yes or no. Yes for « never update the table ».
- OutFile: name of a file (saving) or none. When it is specified the data of the table are saved in the file using the "hexa floating point" format.
- Replace: yes or no. yes stands for «store new value on conflict»
- Tsize: the size of the internal table.

Moreover, exposed variables for runtime autotuning are generated. For foo a memoized function/method:

- `_Memoize_`*Mangling(foo)* when true, the memoization is applied otherwise it is suspended.
- `_alwaysReplace`*Mangling(foo)* when true replace the entry of the table in case of conflict.
- `_FullyOffLine`*Mangling(foo)* set to true to suspend the updating of the table.

3 How to use

Several LARA aspects are proposed for the memoization: the memoization of the mathemathical functions (defined in math.h), Memoization of C user functions and the memoization of C user methods.

3.1 Memoization of Mathematical functions

The more general aspect called `Memoize_MathFunction_ARGS`, shown in Figure 3, is when all the presented memoization parameters are specified by the user.

```
1  aspectdef Memoize_MathFunction_ARGS
2  input
3    target, // name of a math function
4    fileToLoad,// none or file to load to initialize the table
5    FullOffLine,  // yes|no, yes : full offline memoization
6    FileToSave,   // file name to save the table,or none
7    AlwaysReplace, // yes|no. yes:the table is updated in case of collisions
8    precision, // number of bits to delete (0 for int)
9    tsize  // The size of the table.
10 end
11 ...
```

Figure 3: Aspect for the math functions memoization with parameters.

For example, the call to
`Memoize_MathFunction_ARG`('log','none','no','olog.data','yes',0,32768)
specifies the memoization of the `log` function (argument 1)

- without initialization of the table by the content of a file (argument 2),
- the full offline policy of the table is not set (argument 3),
- the resulting table will be saved in a file at the end of the execution (argument 4),
- the table will be updated each time (argument 5). In particular, in case of conflict, the last evaluation is stored,
- the input values will not approximated (argument 6),
- and the size of the internal table is fixed to 2^{15} (argument 7).

Note that the user has no need to specify the number of input arguments and the type of the arguments because they are internally known.

Moreover, the exposed variables

```
1  extern int _alwaysReplacecos, _FullyOffLinecos;
2  int _Memoize_cos = 1; // initialized to true.
```

are provided for runtime autotuning.

The other LARA aspects for mathematical functions are simplified versions of the `Memoize_MathFunctio` aspect:

- `Memoize_MathFunction('log')` Memoization of the `log` function with default parameters. It is equivalent to
  `Memoize_MathFunction_ARG`('log', 'none', 'no', 'none', 'no', 0, 65536)
- `Memoize_MathFunctions(['log', 'sin'])` Memoization of the `log` and `sin` functions with default parameters.
- `Memoize_AllMathFunctions()` Memoization of all the referenced mathematical functions in the application with default parameters.

3.2 Memoization of C user functions

It is quite similar to the previous one. The difference is that the definition of the function is known. There are two aspects:

- `Memoize_Function_ARGS` ('foo', 'none', no, 'res.data', 'yes', 0, 1024) Memoization of the `foo` function with user parameters.
- `Memoize_Function ('foo')` Memoization of the `foo` function with default memoization parameters. It is equivalent to `Memoize_Function_ARGS`('foo', 'none', 'no', 'none','no',0, 65536)

3.3 Memoization of C++ user methods

It is quite similar to the previous one. The difference is that the overlaoding induces by C++ language must be solved. There are four aspects:

- `Memoize_Method_ARGS` ('aClass', 'foo', 'none', 'no', 'res.data', 'yes', 0, 2048) Memoize the aClass::foo method of the aClass class with user parameters.
- `Memoize_Method` ('aClass', 'foo') Memoize the aClass::foo method of the aClass class with default memoization parameters.
- `Memoize_Method_overloading_ARGS`('aClass','foo','float', 2, 'none', 'no', 'none', 'yes', 17, 2048). This aspect is provided for solving the overloading in C++. It specifies the memoization of the aClass::foo method of the aClass class that has 2 inputs arguments of float types.
- `Memoize_Method_overloading`('aClass','foo,'float', 2) This aspect is provided for solving the overloading in C++, it is equivalent to `Memoize_Method_overloading_ARGS` with default memoization parameters.

3.3.1 Using the memoization aspects for a C program

To illustrate the use of the presented memoization aspects, consider the C program shown in Figure 4.

One can see that the functions (left part of Figure 4) `foo` and `tobememoize` are memoizable as well as `log` function (ie they are pure functions without any side effects). To memoize these functions, one can define the Launcher aspect shown in the medium part of Figure 4.

First of all, the memoization LARA package aspects must be imported by

```
1 import antarex.memoi.Memoization;
```

A memoization session must be (line 1) initialized by calling the `Memoize_Initialize()` aspect. It initializes the internal structure of the defined aspects.

The lines 2-4 are examples of calls to the presented aspects for memoization applied to the selected functions.

The memoization session is then ended (line 5) by a call to the `Memoize_Finalize()` aspect. It finalizes the internal structure of the defined aspects.

The effects of the execution of the Launcher aspect is shown on the right part and the bottom of the figure 4.

In the rigth part, the calls to the memoized functions (for example `foo`)? are replaced by a call to the associated wrapper (for example `foo_wrapper`). New includes (`memoization_exposedVars.h`,

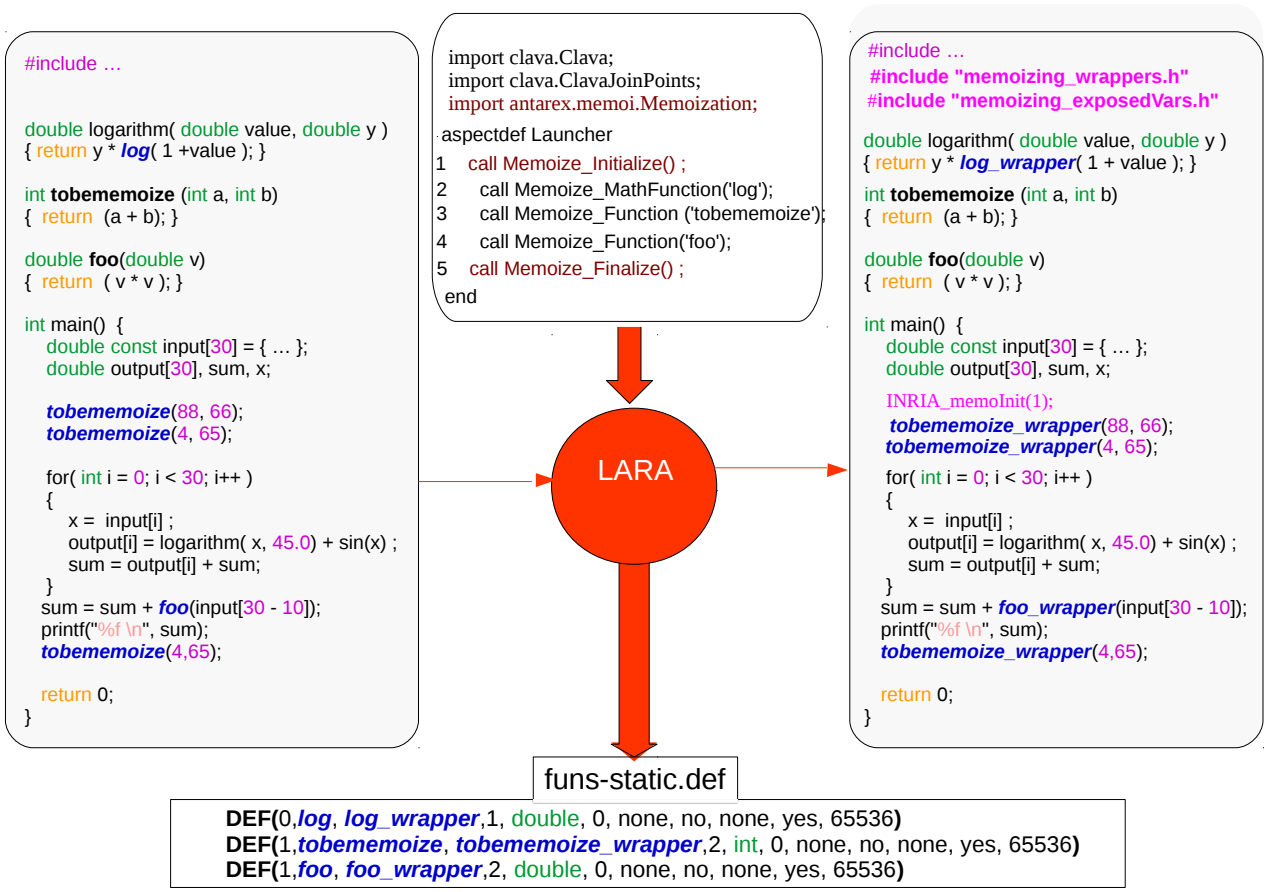

Figure 4: The memoization using LARA

memoization_wrappers.h) are added to the weaved sources. The memoization_exposedVars.h file contains the declarations of the exported variables for runtime autotuning with their initialisations (See Figure 5).

```
#ifndef _MEMOIZING_EXPOSEDVARS_H_
#define _MEMOIZING_EXPOSEDVARS_H_

extern int _alwaysReplacefoo, _FullyOffLinefoo;
int _Memoize_foo = 1;

extern int _alwaysReplacetobememoize, _FullyOffLinetobememoize;
int _Memoize_tobememoize = 1;

extern int _alwaysReplacelog, _FullyOffLinelog;
int _Memoize_log = 1;

 void INRIA_memoInit(int B) {
  _Memoize_foo= B;
  _Memoize_tobememoize= B;
  _Memoize_log= B;
}
#endif
```

Figure 5: Exposed variables declarations.

The memoization_wrappers.h file contains the declarations of the associated wrappers as shown in Figure 6.

```
#ifndef _MEMOIZING_WRAPPERS_H_
#define _MEMOIZING_WRAPPERS_H_

#ifdef __cplusplus
#define __EXTERN__ extern "C"
#else
#define __EXTERN__
#endif

__EXTERN__ double foo_wrapper (double);
__EXTERN__ int tobememoize_wrapper (int , int);
__EXTERN__ double log_wrapper(double);
#endif
```

Figure 6: Declarations of the wrappers.

The second part of the produced code with LARA is the file called "funcstatic.def". It contains the required information to produce the definitions of the wrapper functions that will be used to produce a library. This libray will be linked with the weaved code to produce the final binary. It will be presented after the presentation of the use of the memoization aspects for a C++ program.

3.3.2 Using the memoization aspects for a C++ program

To illustrate the use of the presented memoization aspects on a C++ program, consider the 'toyExample' program in Figure 7. In this code, the main references two objects of different

classes `anObj` of class `Test` and `anObj2` of the class `Test2`. The code of these classes is also shown in Figure 7.

One can observe that the methods `:int Test::function1(int x, int y)` and `int Test2::function1(int x)` are memoizable: they are pure methods and they satisfy the interface constraints expressed above. To memoize these methods, the aspect shown in Figure 8 may be applied.

Note that in this example, the mangling induced by the C++ compiler must be taken into account. The definitions in the file `funs-static.def` (See Figure 9) are generated taking into account this mangling.

```
1  DEF(1,_ZN5Test29function1Ei, _ZN5Test217function1_wrapperEi,1,int,0,none,no
       ,none,no,65536 )
2  DEF(1,_ZN4Test9function1Eii, _ZN4Test17function1_wrapperEii,2,int,17,none,
       no,none,yes,2048 )
3   \end{lstlisting}
```

Figure 9: Declaration of the functions to memoize.

On Figure 10, the weaved code by LARA is presented. Note that for C++, the wrapper is a new method declared in the associated class(See Figure 11).

```
1  #include <iostream>
2  #include <cmath>
3  #include <stdlib.h>
4  #include "Test.h"
5  #include "Test2.h"
6  #include "memoizing_exposedVars.h"
7  using namespace std;
8  int main(int argc, char * argv[]) {
9     INRIA_memoInit(1);
10    int y, x, z;
11    Test anObj;
12    Test2 anObj2;
13    x = 0;
14    int N = 1000;
15    for(int k = 0; k < N; k++)
16     for(int i = 0; i < 1000000; i++) {
17       y = anObj.function1_wrapper(10 + i, 20 + i) + anObj2.function1_wrapper(33);
18       z = anObj.function1_wrapper(10 + i, 20 + i);
19       x = x + y + z + anObj.function1_wrapper(100, 28) + anObj2.function1_wrapper(100);
20    }
21    cout << " x = " << x << endl;
22    return x;
23 }
```

Figure 10: Weaved code of the memoized C++ example.

```
1  #ifndef _TEST_H_
2  #define _TEST_H_
3  class Test {
4   private:
5    int data1;
6    float data2;
7   public:
8    int function1_wrapper ( int , int );
9    int function1(int x, int y);
10   float function2();
11 };
12 #endif
13
14 #ifndef _TEST2_H_
15 #define _TEST2_H_
16 class Test2 {
17  private:
18   int data1;
19   float data2;
20  public:
21   int function1_wrapper ( int );
22   int function1(int x);
23   float function2();
24 };
25 #endif
```

Figure 11: Weaved code (classes) of the memoized C++ example.

3.3.3 Producing the memoization library

To run the application with memoization, a library is produced from the content of the `funs-static.def` file. It requires the memoization package provided at [4]. Ones can import the templates/CMakeList.txt file [5] provided in the distribution and adapt it to the application. Figure 12 shows the modified version of the example.

```
1  PROJECT(MYPROJECT CXX C)
2
3  # require the MEMOIZATION environment.
4  IF(NOT EXISTS "$ENV{MEMOIZATION_ROOT}")
5    MESSAGE("=====================================================")
6    MESSAGE("The variable MEMOIZATION_ROOT is not defined...exiting")
7    MESSAGE("=====================================================")
8    MESSAGE(FATAL_ERROR)
9    RETURN()
10 ENDIF(NOT EXISTS "$ENV{MEMOIZATION_ROOT}")
11
12 SET(MEMOIZATION_ROOT $ENV{MEMOIZATION_ROOT})
13
14 # Binary name of my application.
15 SET(APP_NAME toyExampleCpp)
16
17 # Sources
18 SET(APP_NAME_SRC toyExampleCpp.cpp Test2.cpp  Test.cpp)
19
20 # Libraries
21 SET(LIBS m)
22
23 # Comment this line to have a non dynamical management of the memoization.
24 ADD_DEFINITIONS( -DDYNAMIC_MODE )
25
26 # Uncomment this line to have some statistics about the memoization.
27 # ADD_DEFINITIONS( -DMEMOISTATS)
28
29 # Production of the memoization library (memoize) from funs-static.def
30 INCLUDE(${MEMOIZATION_ROOT}/templates/memoization.cmake)
31
32 ADD_EXECUTABLE( ${APP_NAME} ${APP_NAME_SRC})
33 # Application is linked with the memoize library.
34 TARGET_LINK_LIBRARIES(${APP_NAME} ${LIBS} memoize)
```

Figure 12: the cmake file of the 'toy example'

On this figure,

- The lines 3-10 test if the memoization library is known in the user environment: the shell variable `MEMOIZATION_ROOT` must be set.
- The lines 14-21 are used to defined the components of the user application (source, includes, libraries)
- The line 24 specifies if the memoization will be managed in a dynamical way. When the definition is present, the memoization of functions and methods may be suspended and restarted dynamically. The user can reference the exposed variables in its source.
- The line 27 specifies the production of some statistics about the memoization library. For each memoized function or method, the number of requests, the number of hits and the number of collisions are printed out.
- The line 30 specifies the production of the memoization library, using the `funs-static.def` file (on Linux, the generated file will be `libmemoize.a`). Note that the produced memoization library of the application (memoize) is linked with the user application (line 34).

4 Experimental results

Results of the application of the memoization have been integrated in a paper submitted to the softwareX journal [6]. Overall, the use of our memoization framework allows us to achieve considerable reductions in both execution time and energy consumption.

We give here some other results not published about the application of approximation during the memoization. The figure 13 recalls the principles of the approximation. The approximation parameter, that has sense on the significant, allows to not distinguish very near parameter values of the inputs of the function or method.

To illustrate the impact of the approximation parameter, we have tested on two of the benchmark examples exposed in the softwareX journal:

- **fft** is a Fast Fourier transform implementation extracted from the BenchFFT [7] benchmark suite. It calls the functions `sin` and `cos`. The maximun gain (14.18%) was obtained when the size of the table is 256 and the table is updated each time on collision. Playing with the approximation parameter of the memoization, the gain may be around 20% when this parameter is equal to 32, without any loss of precision in the results.

- **rgb2hsi** is a benchmarking kernel that converts images from RGB model to HSI model. It calls `cos`, `acos`, `sqrt`, and a pure user function. The maximun gain (27.07%) was obtained when the size of the table is 65536 and the table is updated each time on collision. With an approximation parameter fixed to 47, the gain is about 32%, without visible effects on the resulting picture.

5 Conclusion

This chapter presented the framework developed in the ANTAREX project to automatically apply memoization on C/C++ applications. The resulting applications store outputs of pure functions mapped by their inputs. If these pure functions are called with the same inputs, the framework returns the stored value instead of recomputing it.

Hence, this technique may lead to execution time and energy consumption improvements by simply avoiding unnecessary computations in scenarios where critical functions are called with repeating inputs. The results of the application of this technology on several examples show the usefulness of the memoization technique. The advantage of the developed aspects is that the memoization is integrated into the application without requiring user modifications of the source code. The code generated by Clava is then compiled and linked with the associated generated memoization library.

References

[1] Arjun Suresh, Bharath Narasimha Swamy, Erven Rohou, and André Seznec. Intercepting Functions for Memoization: A Case Study Using Transcendental Functions. *ACM Trans. Archit. Code Optim.*, 12(2), 2015.

[2] Arjun Suresh, Erven Rohou, and André Seznec. Compile-Time Function Memoization. In *26th International Conference on Compiler Construction*, Austin, United States, 2017.

[3] Name mangling. `https://en.wikipedia.org/wiki/Name_mangling#C++`.

[4] The memoization package. `https://gforge.inria.fr/projects/memoization`.

[5] Build, Test and Package Your Software With CMake. `https://cmake.org`.

[6] Loïc Besnard, Pedro Pinto, Imane Lasri, João Bispo, Erven Rohou, and João MP Cardoso. A Framework for Automatic and Parameterizable Memoization. *SoftwareX Journal, https://www.journals.elsevier.com/softwarex*, 2018. Manuscript submitted for publication.

[7] Bench FFT. `http://www.fftw.org/benchfft`.

```cpp
// ======  toyExample.cpp ==============================
#include <iostream>
#include <cmath>
#include <stdlib.h>

#include "Test.h"
#include "Test2.h"
using namespace std;

int main (int argc, char *argv[])
{
  int y, x, z;
  Test anObj;
  Test2 anObj2;

  x = 0;
  int N = 1000;
  for (int k = 0; k < N; k++)
    for (int i =0; i < 1000000 ; i++) {
      y  = anObj.function1(10+i, 20+i) + anObj2.function1(33);
      z = anObj.function1(10+i, 20+i);
      x = x + y + z +  anObj.function1(100, 28) + anObj2.function1(100);
    }
  cout << " x = " << x << endl;
  return x;
}
```

```cpp
// ======  with Test2.cpp ========================
#include "Test2.h"

int Test2::function1(int x) {
  this->data1 = x + x;
  return this->data1;
}

float Test2::function2() {
  this->data2 = 3.5;
  return this->data2;
}
```

```cpp
// ======  with Test.cpp ========================
#include <cmath>
#include "Test.h"

int Test::function1(int x, int y) {
  this->data1 = x + y + x*y + x/y + y/x + log(x);
  return this->data1;
}

float Test::function2() {
  this->data2 = 3.5;
  return this->data2;
}
```

```cpp
// ====== Test2.h ========================
  class Test2 {
    private:
      int data1;
      float data2;
    public:
      int function1(int x);
      float function2();
 };
```

```cpp
// ==== Test.h ========================

  class Test {
    private:
      int data1;
      float data2;
    public:
      int function1 ( int x , int y );
      float function2();
  };
```

Figure 7: Code of a 'toy example' C++ program.

```
1  import clava.Clava;
2  import clava.ClavaJoinPoints;
3  import antarex.memoi.Memoization;
4
5  aspectdef Launcher
6     call Memoize_Initialize( );
7          call Memoize_Method_ARGS('Test', 'function1','none', 'no', 'none', '
              yes', 17, 2048);
8          call Memoize_Method('Test2', 'function1');
9          call Memoize_Finalize( );
10 end
```

Figure 8: LARA aspect for the memoization C++ program.

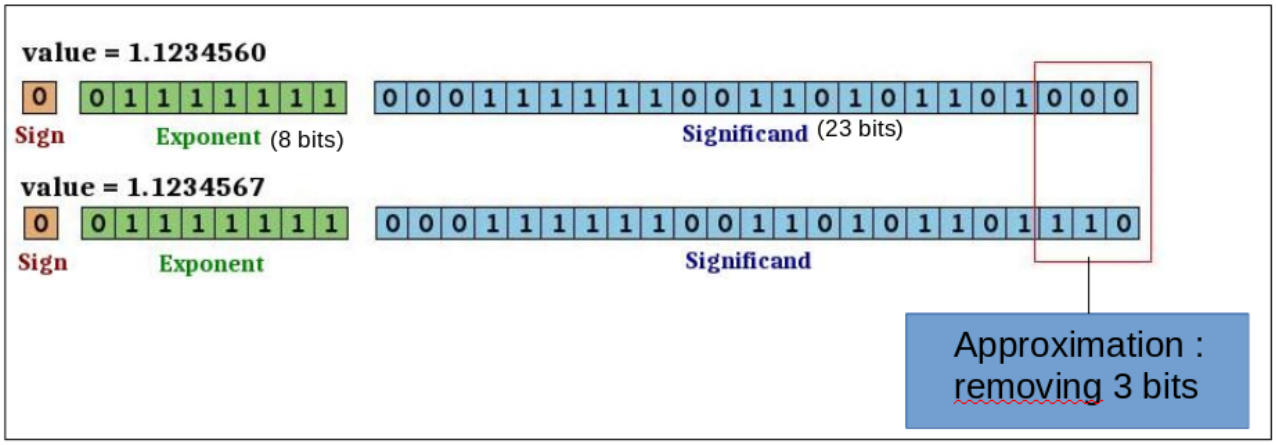

Figure 13: Principles of the approximation.