Chapter 7. LARA Strategies for Loop Splitting

Loïc Besnard

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

Abstract

This chapter presents a technique, called loop splitting, that takes advantage of long running loops to explore different compiling options to optimize the user applications. This technique may be also used to explore different implementation of algorithms. LARA aspects have been developed to apply the technique in a very simple way.

1 Introduction

Compilers rely on hundreds of optimizations to deliver performance [1] [2]. Optimization levels (such as -O3 with the gcc compiler) are highly tuned to generate good code on a set of representative benchmarks. For a single application, however, the heuristics can often be significantly surpassed. Iterative compilation has been proposed in the late 1990's [3] [4] [5] to explore a large domain space in order to find better optimization sequences. The drawback is the need for a large number of executions. We present in this chapter a technique to take advantage of long running loops to explore the impact of several optimization sequences at once, thus reducing the number of necessary runs.

The remainder of this chapter is organized as follows. We describe first the principles of the loop splitting technique, followed by the presentation of the LARA aspects developed in ANTAREX to implement this technique. In a third section, experimental results are presented. The conclusion outlines future directions and concludes our work.

2 Principles of the loop splitting

We rely on a variant of loop peeling which splits a loop into several loops, with the same body, but a subset of the iteration space. New loops execute consecutive chunks of the original loop. We then apply different optimization sequences on each loop independently. Timers around each chunk observe the performance of each fragment.

This technique may be generalized to combine compiler options and different implementations of a function called in a loop. It is useful when, for example, the profiling of the application shows that a function is critical in term of time of execution. In this case, the user must try to find the best implementation of its algorithm.

To explain the principles of this technique, consider the C code shown in Figure 1.

```
1
2
3  void afunction(p1, ...pm) {
4    ...
5    statement_0;
6    for (i=0; i< N; i++) {
7      statement_1;
8      ...
9      statement_n;
10   }
11   statement_n+1;
12   ...
13 }
```

Figure 1: C code with a loop.

To study the effect of 4 sets of options in the internal loop, this code may be transformed by

- the creating of a new software component as shown Figure 2, that defines the computation of the loop but with lower and upper bounds provided as parameters.

```
1
2
3  void INRIA_SPLITTED_LOOP_1(int lb_INRIA_SPLITTED_LOOP, int
       ub_INRIA_SPLITTED_LOOP,...)
4  {
5   for (i=lb_INRIA_SPLITTED_LOOP; i< ub_INRIA_SPLITTED_LOOP; i++) {
6     statement_1;
7     ...;
8     statement_n;
9   }
10 }
```

Figure 2: Loop splitting:loop isolated as a function/method.

- the synthesis of the variables referenced in the body of the original loop: for the parameters of the enclosing function, they are arguments of the created function. For the other ones, they are renamed and declared as global variables. The initialization of such a variable is kept in the body of the original function. This part is not specified in the different figures for legibility reasons.
- the replacing of the loop by a series of blocks as shown in Figure 3, (lines 5-9, lines 11-15, lines 17-21, lines 23-27) with the same structure:
  - the call to the load (`INRIA_SPLITTED_LOOP_1_load()`) of a version of the function compiled with a set of compiler options. This function, based on `dlsym` and `dlopen` functions for Linux system, loads a dynamic shared library and get the address of the symbol associated with the created function.
  - the call to the created function (`INRIA_SPLITTED_LOOP_1()`) with adapted parameters for the lower and upper bounds values. Timers are added around this call to get its execution time.

```
1
2  statement_0;
3  if (INRIA_SPLITTED_LOOP_1_LEARNING ) {
4      INRIA_SPLITTED_LOOP_1_SELECTED= 0;
5     // Running the first version
6     INRIA_SPLITTED_LOOP_1_load(libV0);
7     t0_begin = getCurrentTime();
8     INRIA_SPLITTED_LOOP_1(0, N/4);
9     t0 = getCurrentTime() - t0_begin;
10
11    // Running the second version
12    INRIA_SPLITTED_LOOP_1_load(libV1);
13    t1_begin = getCurrentTime();
14    INRIA_SPLITTED_LOOP_1(N/4)+1, 2*(N/4);
15    t1 =  getCurrentTime() - t1_begin;
16
17    // Running the third version
18    INRIA_SPLITTED_LOOP_1_load(libV2);
19    t2_begin = getCurrentTime();
20    INRIA_SPLITTED_LOOP_1( 2*(N/4)+1, 3*(N/4));
21    t2 = getCurrentTime() - t2_begin;
22
23    // Running the fourth version
24     INRIA_SPLITTED_LOOP_1_load(libV3);
25    t3_begin = getCurrentTime();
26    INRIA_SPLITTED_LOOP_1(3*(N/4)+1, N);
27    t3 = getCurrentTime() - t3_begin;
28
29    INRIA_SPLITTED_LOOP_1_LEARNING=0; // end learning
30  }
31  else {
32    if (! INRIA_SPLITTED_LOOP_1_SELECTED) {
33       INRIA_SPLITTED_LOOP_1_SELECTED = 1;
34       // Selecting the best version
35       int X  = INRIA_SPLITTED_LOOP_BEST(...);
36       // Loading the selected one
37        INRIA_SPLITTED_LOOP_1_load(libV<X>);
38    }
39    // Running for all the index values
40    INRIA_SPLITTED_LOOP_1(0, N);
41  }
42
43 statement_n+1;
```

Figure 3: Loop splitting: the chunks.

Two global variables are generated to manage the execution of the application:

- the first one, `INRIA_SPLITTED_LOOP_1_LEARNING`, when it is `true` induces the running of the application in "learning" mode: the execution of the created chunks with the execution time.
- The second one, `INRIA_SPLITTED_LOOP_1_SELECTED`, when it is `false`, induces (lines 32-37, figure 3) the selection of the "best" version, and the loading of the selected library.

When the variable `INRIA_SPLITTED_LOOP_1_LEARNING` is `false` and the variable `INRIA_SPLITTED_LOOP_`

`true` the loop is executed with the selected library on the original bounds of the loop (line 40, figure 3).

Note that when the loop is inside a method, a new method is created (instead a function) and added to the associated class. In this case, the mangling must be used in the `INRIA_SPLITTED_LOOP_1_load()` function and the call to the method must be modified to add the object as the first parameter of this call.

The presented technique has been implemented by the definition of LARA aspects in the ANTAREX project. It is restricted currently to Linux systems and for the classical C loop form:

```
1 for ( initial; test ; iteration) {...}
```

where `initial` is the first bound for the index of the loop (for examples, `i=0` or `i=999`), `test` is the condition for finalyizing the loop (for examples, `i<777` or `i > 0`) and `iteration` is the increment or decrement of the index (for examples `i++`, `i--`, `i=i+3`,...).

3 How to use

To apply the presented technique, a pragma (`#pragma SPLITLOOP N`) must be inserted in the source of the application by the user, where `N` is the number of chunks to produce. This pragma must be assigned to the line before the loop as shown line 3 in Figure 4.

```
1
2
3 #pragma SPLITLOOP 8
4 for (int pose_index = 0; pose_index < 256; ++pose_index) {
5  // get the configuration to apply to the best fragments
6  std::vector<double> configuration =...
7  ...
8 }
```

Figure 4: Loop splitting principles: using pragma.

A loop splitting LARA session is shown in Figure 5.

```
1  import lara.Io;
2  import clava.Clava;
3  import clava.ClavaJoinPoints;
4  import antarex.split.splitLoopAspects;
5
6  aspectdef Launcher
7       call splitLoops_initialize();
8       call splitLoops();
9       call splitLoops_finalize();
10 end
```

Figure 5: Loop splitting principles: launcher aspect.

First of all, the loop splitting LARA package must be imported by

```
1 import antarex.split.splitLoopAspects;
```

Such a session must be (line 7) initialized by calling the `slitLoops_Initialize()` aspect. It initializes the internal structure of the defined aspects.

The line 8 applies the loop splitting to the attributed loops with the specified pragma. It produces for each attributed loop the following components:

- `INRIA_SPLITTED_LOOP_XX_code.cpp` that defines the created function/method
- `INRIA_SPLITTED_LOOP_XX_load.h` that defines the dynamical loader of the libraries.

where XX denotes an integer associated with the loop to ensure the unicity of the symbols.

An another file called `INRIA_SPLIT_LOOP_BEST_CODE.h` is also generated. It defines the function that selects the best choice.

The session is then ended (line 9) by a call to the `splitLoops_Finalize()` aspect. This aspect finalizes the internal structure of the defined aspects.

4 Selected experimental results

As a proof of concept, the loop splitting technique has been applied to an ANTAREX Use Case (Computer Accelerated Drug Discovery).

A example of trace of the execution of this application is shown in Figure 6. The pragma was fixed to 8 and the options for each library are shown in figure 7.

```
1  Docking on [ 1d3h 1d3h 1d3h ] protein pockets
2    Target protein pocket: "1d3h"
3  >>>>>>  Loading libINRIA_SPLITTED_LOOP_1_1.so library
4   MODE_LEARNING SPLIT 1 took 6741703 ms to run.
5  >>>>>>  Loading libINRIA_SPLITTED_LOOP_1_2.so library
6   MODE_LEARNING SPLIT 2 took 6150785 ms to run.
7  >>>>>>  Loading libINRIA_SPLITTED_LOOP_1_3.so library
8   MODE_LEARNING SPLIT 3 took 6498927 ms to run.
9  >>>>>>  Loading libINRIA_SPLITTED_LOOP_1_4.so library
10  MODE_LEARNING SPLIT 4 took 6394542 ms to run.
11 >>>>>>  Loading libINRIA_SPLITTED_LOOP_1_5.so library
12  MODE_LEARNING SPLIT 5 took 6302113 ms to run.
13 >>>>>>  Loading libINRIA_SPLITTED_LOOP_1_6.so library
14  MODE_LEARNING SPLIT 6 took 6469709 ms to run.
15 >>>>>>  Loading libINRIA_SPLITTED_LOOP_1_7.so library
16  MODE_LEARNING SPLIT 7 took 6183150 ms to run.
17 >>>>>>  Loading libINRIA_SPLITTED_LOOP_1_8.so library
18  MODE_LEARNING SPLIT 8 took 6431997 ms to run.
19 Elapsed time for a pocket = 51.2135 seconds
20   Target protein pocket: "1d3h"
21 >>>>>>  Loading libINRIA_SPLITTED_LOOP_1_2.so library
22 Elapsed time for a pocket = 49.6977 seconds
23
24 Target protein pocket: "1d3h"
25 Elapsed time for a pocket = 49.7452 seconds
```

Figure 6: Trace of a run with loop splitting.

```
// Implicit option: -march=native
Lib 1 "-O3 "
Lib 2 "-O3 -fpeel-loops -ffast-math "
Lib 3 "-O3 -funroll-loops -ffast-math "
Lib 4 "-O3 -funroll-all-loops -ffast-math "
Lib 5 "-O3 -fomit-frame-pointer -funroll-all-loops          -ffast-math -
   mfpmath=sse -msse2 "
Lib 6 "-O3 -funroll-loops -fno-exceptions -fwrapv           -funsafe-math-
   optimizations "
Lib 7 "-O3 -funroll-loops -fpeel-loops -fno-exceptions      -fwrapv -funsafe
   -math-optimizations "
Lib 8 "-O3 -fpeel-loops "
```

Figure 7: Associated options of the libraries.

The tests have been performed on an Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz. In this example, the algorithm has been applied 3 times on the same inputs: the first one is used to select the best library (learning mode), the second/third are for proving the usefulness of the technique.

On this example, the application of the loop splitting technique produces a gain in term of time of execution. For example, the gain for computing the same number of iterations between the use of the first library, compiled with the "classical" `"-O3 -march=native"` options, and the selected one (library 2 compiled with the `"-O3 -march=native -funroll-all-loops -ffast-math"` is around 8.8%.

The gain compared with the original version (ie without loop splitting) compiled with `"-O3 -march=native"` options is around 4%.

As expressed in section 2, the exposed technique may be generalized to combine compiler options and different implementations of a function called in a loop. The figure 8 shows the trace of the same application in which a function called in the loop is inlined.

```
1
2  Docking on [ 1d3h 1d3h 1d3h ] protein pockets
3    Target protein pocket: "1d3h"
4  >>>>>>  Loading libINRIA_SPLITTED_LOOP_1_1.so library
5   MODE_LEARNING SPLIT 1 took 6404742 ms to run.
6  >>>>>>  Loading libINRIA_SPLITTED_LOOP_1_2.so library
7   MODE_LEARNING SPLIT 2 took 5721446 ms to run.
8  >>>>>>  Loading libINRIA_SPLITTED_LOOP_1_3.so library
9   MODE_LEARNING SPLIT 3 took 5792065 ms to run.
10 >>>>>>  Loading libINRIA_SPLITTED_LOOP_1_4.so library
11  MODE_LEARNING SPLIT 4 took 5676159 ms to run.
12 >>>>>>  Loading libINRIA_SPLITTED_LOOP_1_5.so library
13  MODE_LEARNING SPLIT 5 took 5760976 ms to run.
14 >>>>>>  Loading libINRIA_SPLITTED_LOOP_1_6.so library
15  MODE_LEARNING SPLIT 6 took 5801540 ms to run.
16 >>>>>>  Loading libINRIA_SPLITTED_LOOP_1_7.so library
17  MODE_LEARNING SPLIT 7 took 6001114 ms to run.
18 >>>>>>  Loading libINRIA_SPLITTED_LOOP_1_8.so library
19  MODE_LEARNING SPLIT 8 took 6250686 ms to run.
20 Elapsed time for a pocket = 47.4565 seconds
21   Target protein pocket: "1d3h"
22 >>>>>>  Loading libINRIA_SPLITTED_LOOP_1_4.so library
23 Elapsed time for a pocket = 45.8343 seconds
24   Target protein pocket: "1d3h"
25 Elapsed time for a pocket = 45.8096 seconds
```

Figure 8: Trace of a run with Loop splitting.

In this case, the gain compared with the original version compiled with `"-O3 -march=native"` options is around 11.60%.

5 Conclusion

This chapter presented the framework developed in the ANTAREX project to automatically apply the loop splitting. The advantage of the developed aspects is that the loop splitting is integrated into the application in a very simple way on the source code of an application. The goal of the technique is to find the best combination of compiler options for optimizing the application when a loop is used a very number of times, as it has been shown on a concrete example.

This work should be generalized to other operating systems and to the other forms of the loop iterations of C++ languages.

References

[1] Aho, Sethi, and Ullman. *Compilers, Principles, Techniques, and tools.* Pearson Education, 2007. second edition.

[2] Muchnick Steven S. *Advanced Compiler Design and Implementation.* Elsevier, 2014.

[3] David F Bacon, Susan L Graham, and Oliver J Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys (CSUR)*, 26(4):345–420, 1994.

[4] François Bodin, Toru Kisuki, Peter Knijnenburg, Mike O'Boyle, and Erven Rohou. Iterative compilation in a non-linear optimisation space. In *Workshop on Profile and Feedback-Directed Compilation*, 1998.

[5] Arif Ap, Kévin Le Bon, Byron Hawkins, and Erven Rohou. FITTCHOOSER: A dynamic feedback-based fittest optimization chooser. In *16th International Conference on High Performance Computing & Simulation (HPCS 2018)-Special Session on Compiler Architecture, Design and Optimization*, page 8, 2018.