APOLLO User Guide

Automatic speculative POLyhedral Loop Optimier
January 29th 2017

APOLLO Crew

Table of Contents

1 Overview **1**

2 Usage **2**

3 Runtime Options **3**

3.1 Backdoor 3
3.2 Chunk Size for Profiling Phases 3
3.3 Chunk Size for Optimized Phases 3
3.4 Optimization Flags 4
3.5 Predict All Accesses 4

4 Profiling Mode **5**

4.1 Polyhedral Phases Detector 5
4.2 Reindex Polyhedral Phases Detector 6
4.3 Profiling Frequency 6

1 Overview

APOLLO is a runtime automatic parallelizer of loops of any kind (for, while or do-while loops) whose memory references can also be of any kind (arrays, variables, linked lists, etc. through indirections or pointers). APOLLO implements a speculative strategy that allows to take advantage, at runtime, of loop parallelizing and optimizing techniques based on the polyhedral model. Loop nests that may be optimized and parallelized by APOLLO exhibit runtime phases where the sequence of memory addresses referenced by each memory instruction match a linear or quasi-linear function.

2 Usage

A specific pragma, `#pragma apollo dcop`, where `dcop` stands for *dynamic control part*, allows a user to mark in a source code some loop nests of any kind (for, while or do-while nested loops), in order to be handled by the speculative parallelization process of APOLLO, to take advantage of the underlying multi-core processor architecture. The unique restrictions are:

- The target loops must not contain any instruction performing dynamic memory allocation, although dynamic allocation is obviously allowed outside the target loops.
- Since Apollo does not handle inter-procedural analyses, the target loops should not generally contain any function invocation. However, a called function may be inlined in some cases, thus annihilating this issue.

For example:

```
...
#pragma apollo dcop
{
   while (i < N && B[i] < 100) {
      for (j = 0; j < M; ++j) {
         A[B[i][j] += C[B[j]];
      }
   }
}
```

After installation, APOLLO may be invoked using one of both following commands, for compiling the target source code:

- apolloc for C programs;
- apolloc++ for C++ programs.

Additionally, since APOLLO's static compiler is based on LLVM/Clang, any LLVM/Clang option may be used.
For example:

```
apolloc -O3 my_program.c
```

The produced executable file is a fat binary containing invocation to APOLLO's runtime function, as well as important data such as static analysis information or code-bones in LLVM intermediate representation form. The generated executable file may be launched by the user in the standard way. The target loop nests will then be automatically handled by the speculative parallelization process of APOLLO. Additionally, some runtime options that are described below, may be used.

3 Runtime Options

All runtime options may by set by using some dedicated environment variables.

3.1 Backdoor

APOLLO provides a powerful backdoor mechanism allowing to get deep knowledge of is going on inside APOLLO during an execution. This backdoor mechanism is enabled by defining the `APOLLO_BACKDOOR` environment variable as following:

```
APOLLO_BACKDOOR="<output_option> <todo_option>"
```

`<output_option>` can be:

`--stdio` Enable outputs printing on stdout

`--file=FILE_PATH`
Print outputs in the given file

`--tcp` Send outputs to localhost on port 22222

`<todo_option>` can be:

`--step` Print APOLLO runtime phases

`--info` Print execution information in details, as for example, the prediction model, linear functions, tubes etc.

`--info_all`
Print execution information in even more details, as for example, the recorded profiling events

`--trace` Print trace information

`--time` Print starting and ending time stamps of events

For example:

```
APOLLO_BACKDOOR="--stdio --step" ./my_program
```

3.2 Chunk Size for Profiling Phases

APOLLO allows to configure the size of the profiling phases which default value is 16:

```
APOLLO_INSTR_CHUNK_SIZE=<value>
```

For example:

```
APOLLO_INSTR_CHUNK_SIZE=32  ./my_program
```

3.3 Chunk Size for Optimized Phases

APOLLO allows to configure the size of the optimized phases which default value is 1024:

```
APOLLO_OPT_CHUNK_SIZE=<value>
```

For example:

```
APOLLO_OPT_CHUNK_SIZE=512  ./my_program
```

3.4 Optimization Flags

APOLLO provides several options allowing to configure how the target loop nests are optimized. These options are controlled by defining the `APOLLO_BONES` environment variable as following:

```
APOLLO_BONES="<option> <option> ..."
```

`<option>` can be:

`--control`
: CLooG option. Optimize the generated code for control overhead (disabled by default).

`--delinearize=yes|no|force`
: Enable/disable/force de-linearization of linear memory references (yes by default). Warning: force is unsafe.

`--fuse=max|smart|no`
: Pluto option. Switch between maximal fusion, smart fusion, or no-fusion (no by default).

`--identity`
: Pluto option. Use the identity transformation only (disabled by default). If this option is used, APOLLO will only attempt to parallelize the target loop nests by slicing each outermost loop into parallel threads.

`--islsolve`
: Pluto option. Set isl has ILP solver instead of PiP (disabled by default).

`--no-split`
: Do not split the verification code from the rest of the computation (disabled by default).

`--rar`
: Pluto option. Consider read-after-read dependencies (disabled by default).

`--simplify`
: Merge code bones as much as possible (disabled by default).

`--tile|--notile`
: Force/disable tiling instead of using APOLLO's internal heuristic for tiling activation or not.

`--tubewidth=x`
: Increase the tube width by multiplying de profiled width by x (1 by default).

`--unroll`
: Pluto option. Set unroll option (disabled by default).

For example:

```
APOLLO_BONES="--rar --tile"   ./my_program
```

3.5 Predict All Accesses

By default APOLLO stops creating a prediction model as soon as it encounters a non predictable scalars, memory access or loop bounds. For debug purposes it can be useful to still predict everything. To override the default behavior, use the following environment variable:

```
APOLLO_PREDICT_ALL=1
```

4 Profiling Mode

APOLLO can also be used as a profiler. In this case, instead of optimizing the code, APOLLO gathers and reports information of the target loop nests. To ease the integration of different types of profiling, APOLLO provides a flexible mechanism to develop profiling plugins.
APOLLO profiling mode is enabled by defining the `APOLLO_PROFILING` environment variable as following:

```
APOLLO_PROFILING="<--profiling_plugin>"
```

`<profiling_plugin>` can be:

`--polyhedral_phases_detector`
: The polyhedral phases detector plugin.

`--polyhedral_phases_detector_full`
: The polyhedral phases detector plugin with more detailed reporting.

`--redindex_polyhedral_phases_detector`
: The reindex polyhedral phases detector plugin. Compared to the polyhedral phases detector plugin, this plugin tries to reindex memory accesses in case of non polyhedral behaviour and then try again to build a polyhedral model.

`--redindex_polyhedral_phases_detector_full`
: The reindex polyhedral phases detector plugin with more detailed reporting.

4.1 Polyhedral Phases Detector

The phases detector plugin computes and reports information about the different polyhedral phases observed during an execution. A phases can be:

- Linear
- Linear with tubes
- Non polyhedral

For all types of phases, the polyhedral phases detector plugin reports the lower bound and the upper bound of the phase (in terms of the outer most loop iterator). For both linear and linear with tubes phases, the plugin also reports the OpenSCoP objects before and after polyhedral optimization.
The polyhedral profiling plugin is enabled as following:

```
APOLLO_PROFILING="--polyhedral_phases_detector" ./my_program
```

At the end of the execution, the plugin reports phases information on the standard output. If this information is saved into a file, it can be further processed by the `polyhedral_phases_detector_parse_output.py` script to get higher level information about phases.
For example

```
APOLLO_PROFILING="--polyhedral_phases_detector" ./my_program > output
$APOLLO_BIN_PATH/polyhedral_phases_detector_parse_output.py ./output
```

The result is the following for the matrix multiply example provided along with the APOLLO distribution:

```
Phase 0 from 0 to 96 linear
  Chlore stdout:
```

```
  interchange([0,0,0,0], 2, 3, 0);
Chlore stderr:
  <nothing on stderr>
loop 1 is parallel
```

The profiler indicates that the execution of matrix multiply is made of a single linear phase from iterations 0 to 96 of the outer most loop in the loop nest. In standard execution mode, for this single linear phase, APOLLO would have performed a loop interchange transformation and a parallel execution of the loop at depth 1 of the target loop nest.

4.2 Reindex Polyhedral Phases Detector

The reindex polyhedral phases detector plugin behaves exactly as the polyhedral phases detector plugin described in the previous section. The only difference concerns phases that are not linear. For such phases, this plugin will try to re-index memory accesses before looking for a linear or tube model.

The reindexing step consists in assigning a new address to every memory access. To that end, original memory adresses are re indexed by adresses starting at 0 and incremented by one every time a new address is encountered.

4.3 Profiling Frequency

In profiling mode, APOLLO profiles by default all the iterations of the outer most loop. Nevertheless, this behaviour can be changed to profile less iterations in case the complete profiling is to costly. For that, the following environment variable is provided profiled.:

```
APOLLO_PROFILING_FREQUENCY=<value>
```

For example:

```
APOLLO_PROFILING_FREQUENCY=100
```

This tells APOLLO to run a profiling chunk of size APOLLO_INSTR_CHUNK_SIZE every 100 iterations of the outermost loop. As a consequence, APOLLO_PROFILING_FREQUENCY must be greater than APOLLO_INSTR_CHUNK_SIZE.