FortLab

Release 1.1.0

Youngsung Kim

Oct 04, 2022

CONTENTS:

Welcome to the Kernel Extraction and Tool Development Framework for Fortran Applications.

FortLab is a python framework on that users can create a kernel extraction and analysis tools for Fortran applications. Kernel is a stand-alone software extracted from a large original software. In general, kernel is easier to use than its original software due to its reduced size and complexity. In addition to kernel extraction, FortLab exposes key capabilities related to kernel extraction and analysis: 1) Fortran Source Code Analysis, 2) Compiler option collection, 3) Source code modification, and 4) Data generation for kernel execution. Using these features, FortLab users can create their own tool.

CHAPTER

ONE

GETTING-STARTED

With FortLab, users can extract a stand-alone kernel from a Fortran program. In addition, they can create their own kernel-based analysis tool. To use it, FortLab should be installed on the system where the original Fortran program is compiled and executed.

1.1 Installation

The easiest way to install fortlab is to use the pip python package manager.

```
>>> pip install fortlab
```

You can install fortlab from github code repository if you want to try the latest version.

```
>>> git clone https://github.com/grnydawn/fortlab.git
>>> cd fortlab
>>> python setup.py install
```

Once installed, you can test the installation by running the following command.

```
>>> fortlab --version
fortlab 0.1.15
```

1.2 Requirements

- Linux OS
- Python 3.5+
- Make building tool(make)
- C Preprocessor(cpp)
- System Call Tracer(strace)
- Compiler(s) to compile your Fortran application

1.3 Using Fortlab built-in apps

With FortLab, you can collect information about building and running a Fortran application or can instrument original source code to generate runtime information such as kernel timing. This section briefly explains how FortLab works by showing an example of collecting compiler command line flags per each source files that are compiled during the application build ("compileroption").

"compileroption" app collects compiler flags from any build system including Makefile, Cmake, or any custom build system.

To demonstrate, we created a simple Makefile that runs gfortran shown below. However, you can change the content of Makefile including compiler command to fit your needs:

<Makefile>

```
compile:
        gfortran -O3 -DNELEMS=10 fortex1.F90
```

Following Linux command runs fortlab with compileroption app to collect compiler flags from running above Makefile. It is assumed that fortlab is installed on the system as explained above.

<fortlab Linux command>

```
>> fortlab compileroption "make compile" --savejson compopts.json
```

Following json file is generated from running the compileroption app.

<compopts.json>

```
{
    "/autofs/nccs-svm1_home1/grnydawn/repos/github/fortlab/examples/fortex1.F90": {
        "compiler": "/usr/bin/gfortran",
        "include": [],
        "macros": [
            [
                "NELEMS",
                "10"
            ]
        ],
        "openmp": [],
        "options": [
            "-O3"
        ],
        "srcbackup": [
            "/autofs/nccs-svm1_home1/grnydawn/repos/github/fortlab/examples/backup/src/0"
        ]
    }
}
```

"srcbackup" is a list of backup copies of the source files used during the compilation. This feature may be needed in the case that a build system dynamically generates and deletes source files at compile time.

To see more examples that uses other FortLab apps, please see *FortLab Built-in Apps*.

1.4 Building and running a custom Fortlab apps

You can create and run your own Fortlab app by optionally using one or more Fortlab builtin apps. Please see *Building and running Fortlab custom apps* for more details.

CHAPTER

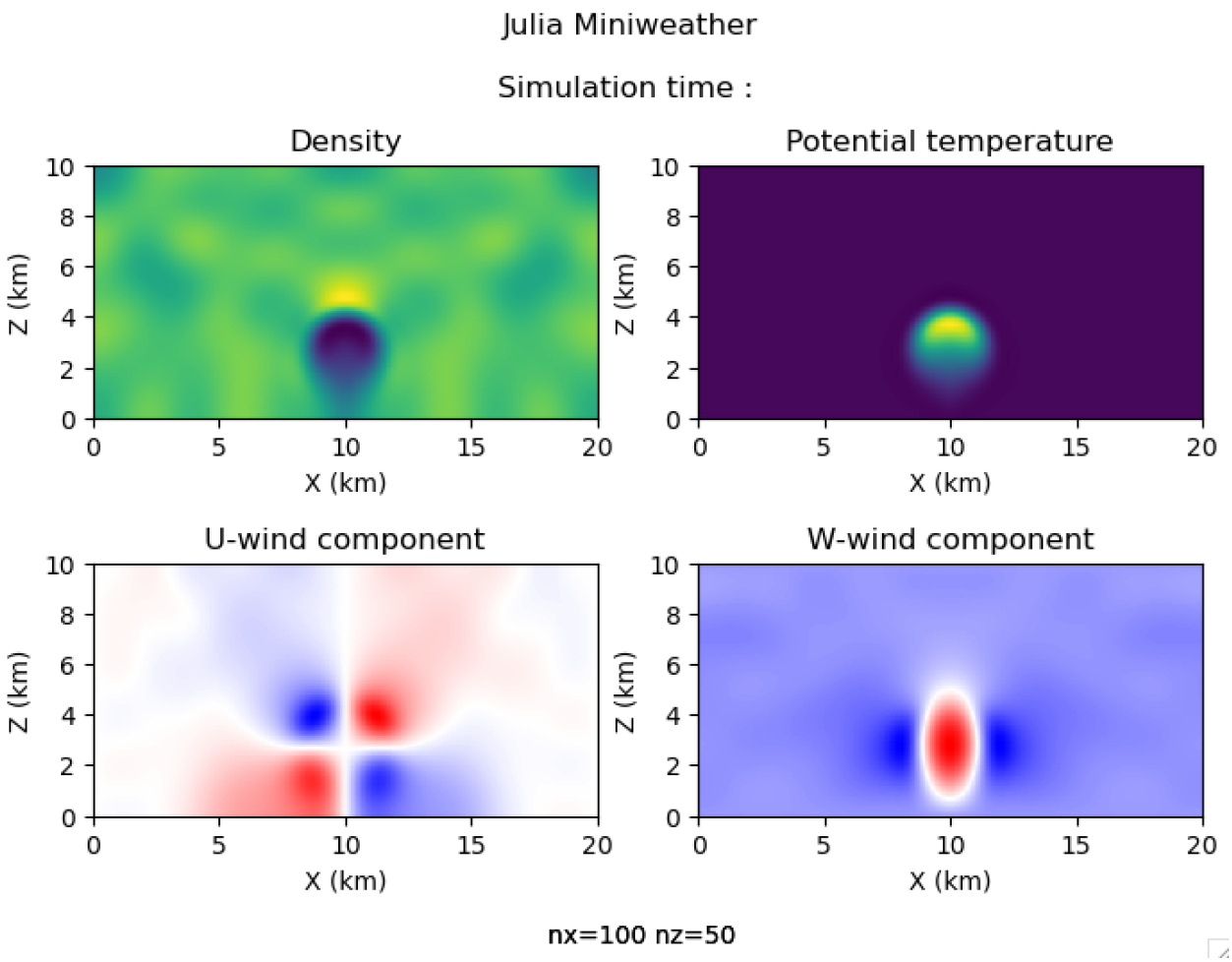
TWO

FORTLAB BUILT-IN APPS

Fortlab is consist of multiple applications that can be assembled together to generate a kernel-based software tools. As of this version, there are following apps in Fortlab.

- compileroption : compiles the target application and collect compiler options per each compiled source files.
- timinggen : generates the elapsed time of the specified kernel region in JSON file, per every MPI ranks, every OpenMP threads(if any), and every invocation of the code regions
- resolve : generates cross-referece information of all Fortran names used in the specified kernel region directly as well as indirectly.
- kernelgen : generates the kernel source files and data files to drive the extracted kernel.
- vargen : generates source files that contains the cross-referece information of all Fortran names used in the specified kernel region

To explain Fortlab apps, we will use miniWeather fortran application. miniWeather is an accelerant app mainly developed by Dr. Matthew Norman. miniWeather simulates weather-like flows for training in parallelizing accelerated HPC architectures. Please see here for the source code of miniWeather Fortran MPI version that is used in this examples.



You can find all the commands used in the following examples in Fortlab Github repository.

2.1 compileroption app

Compiler options are important information to understand how a program source code is translated to a binary code. For example, macros in compiler command-line are used for many large applications. In addition, using the exact compiler options that are used to compile the original software is crucial to improve the representativeness of the generated kernel.

FortLab uses Linux "strace" tool that traces system calls and signals. FortLab executes a Linux command that builds the original software under "strace" and parses the output from strace by tracing "execve" system call. The "strace" com- mand and options used in the app are shown in Listing 9. As a result of this app, developers can get all macro definitions and include paths per every source file used in the compilation. A JSON file can be generated from this app so that developers can use it for later use. For example, "re- solve" app in Section 2.4.1 reads this JSON file through the "–compiler-info" option.

```
>> strace f s 100000 e trace=execve q v  /bin/sh c "compilecommand"
```

"compile-command" is where to put the command-line string for compiling the original software. The "-e" option of "strace" sets to collect only "execve" system calls that are used by compiler invocations.

2.1.1 Example

To explain Fortlab compileroption app, we will use Fortran MPI version of miniWeather as introduced in *FortLab Built-in Apps* section.

To collect compiler flags from compilation of miniWeather.F90, we ran following fortlab command with compileroption subcommand. Following shows the command line that compiles miniWeather in a Makefile. You can find the entire code of the Makefile at https://github.com/grnydawn/fortlab/blob/master/examples/miniWeather/Makefile.

```
INCLUDES := -I...
LIBS := -L...
MACROS := -D_NX=${NX} -D_NZ=${NZ} -D_SIM_TIME=${SIM_TIME} \
          -D_OUT_FREQ=${OUT_FREQ} -D_DATA_SPEC=${DATA_SPEC}

FORTSRC := miniWeather_mpi.F90
F_FLAGS := ${INCLUDES} ${LIBS} ${MACROS} -h noacc,noomp
FC := ftn

miniweather_fort.exe: ${FORTSRC}
    ${FC} ${F_FLAGS} -o $@ $<
```

Following Linux shell command runs Fortlab compileroption app.

```
fortlab compileroption "make miniWeather_fort.exe" --savejson miniWeather_compopts.json
```

"fortlab" is a main command to drive its subcommands. In above example, "compileroption" sub-command is used to collect compiler flags. The actual command for compilation is shown inside of double-quotation marks. The compiler flags can be collected from child processes. For example, this example uses a Makefile. You can optionally save the result to Json file using "–savejson" sub option.

Once the above command runs with success, "miniWeather_compopts.json" file will be created. The content of the json file is shown below.

```
{
    "/.../fortlab/examples/miniWeather/miniWeather_mpi.F90": {
        "compiler": "/opt/cray/pe/craype/2.7.15/bin/ftn",
        "include": [
            "/include"
        ],
        "macros": [
            [
                "_NX",
                "100"
            ],
            [
                "_NZ",
                "50"
            ],
            [
                "_SIM_TIME",
                "10.0"
            ],
            [
                "_OUT_FREQ",
                "10.0"
```

(continues on next page)

(continued from previous page)

```
            ],
            [
                "_DATA_SPEC",
                "1"
            ]
        ],
        "openmp": [],
        "options": [
            "-h",
            "noacc,noomp"
        ],
        "srcbackup": [
            "/.../fortlab/examples/miniWeather/backup/src/0"
        ]
    }
}
```

As you can see the details of compiler and compiler options are saved in Json file. "srcbackup" is a list of backup copies of the source files used during the compilation. This feature may be needed in the case that a build system dynamically generates and deletes source files at compile time. The information in this Json file may be further used for another applicationp. In case of kernel extraction, the information in this Json file is used to analyze source files with proper include paths and macro definitions.

2.1.2 Usage

compileroption app is invoked as a subcommand of fortlab command. You may first check the usage of fortlab command explained in a *overview section* if you are not familiar with fortlab command.

usage: fortlab-compileroption [-h] [–version] [–cleancmd CLEANCMD] [–workdir WORKDIR] [–savejson SAVEJSON] [–backupdir BACKUPDIR] [–verbose] [–check] build command

positional arguments:
build command Software build command

optional arguments:

-h, --help show this help message and exit

--version show program's version number and exit

--cleancmd CLEANCMD CLEANCMD is a Linux shell command that clear all the object and other intermittent files. You may wrap the command with double or single qutation marks if there exist spaces in the command.

--workdir WORKDIR Any output files will be crated in WORKDIR

--savejson SAVEJSON Collected compiler options will be saved in a JOSN file of SAVEJSON

--backupdir BACKUPDIR To support the case that a build-system generates new source files during the build phase but delete before completing compilation, this app saves all the source files used in the build in BACKUPDIR

--verbose show compilation details

--check check strace return code

This app may feed-forward following data to next app:

data (type=any) The collected compiler options can be used as an input data to next Fortlab app without saving as a file. If an app is linked as a next app of this compileroption app, the linked app can use the compiler flags with an input argument name of "data".

2.2 timinggen app

This app instruments the original code to insert timing generation code. The timing collection measures the elapsed time between the beginning and the end of the specified kernel region per every execution from all threads and processes. For example, in case of MPI and OpenMP enabled software, the total number of measurements is the product of the number of invocations, the number of OpenMP threads, and the number of MPI ranks.

2.2.1 Example

To explain Fortlab compileroption app, we will use Fortran MPI version of miniWeather as introduced in *FortLab Built-in Apps* section.

To collect timing data from compilation of miniWeather.F90, we ran following fortlab command with timinggen sub-command. Following shows the command line that compiles miniWeather in a Makefile. You can find the entire code of the Makefile at https://github.com/grnydawn/fortlab/blob/master/examples/miniWeather/Makefile.

```
INCLUDES := -I...
LIBS := -L...
MACROS := -D_NX=${NX} -D_NZ=${NZ} -D_SIM_TIME=${SIM_TIME} \
          -D_OUT_FREQ=${OUT_FREQ} -D_DATA_SPEC=${DATA_SPEC}

FORTSRC := miniWeather_mpi.F90
F_FLAGS := ${INCLUDES} ${LIBS} ${MACROS} -h noacc,noomp
FC := ftn

miniweather_fort.exe: ${FORTSRC}
    ${FC} ${F_FLAGS} -o $@ $<
```

Before running fortlab command, we need to specify where to measure the timing in the source code. In this example, we specified the region for timing generation in the subroutine of compute_tendencies_z as shown below.

```
!$kgen begin_callsite tend_z

    !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
    !! TODO: THREAD ME
    !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
    !Compute fluxes in the x-direction for each cell
    do k = 1 , nz+1
      do i = 1 , nx
        !Use fourth-order interpolation from four cell averages to compute the value at␣
↪the interface in question
        do ll = 1 , NUM_VARS
          do s = 1 , sten_size
            stencil(s) = state(i,k-hs-1+s,ll)
          enddo
          !Fourth-order-accurate interpolation of the state
          vals(ll) = -stencil(1)/12 + 7*stencil(2)/12 + 7*stencil(3)/12 - stencil(4)/12
```

(continues on next page)

(continued from previous page)

```
          !First-order-accurate interpolation of the third spatial derivative of the␣
↪state
          d3_vals(ll) = -stencil(1) + 3*stencil(2) - 3*stencil(3) + stencil(4)
        enddo

        !Compute density, u-wind, w-wind, potential temperature, and pressure (r,u,w,t,p␣
↪respectively)
        r = vals(ID_DENS) + hy_dens_int(k)
        u = vals(ID_UMOM) / r
        w = vals(ID_WMOM) / r
        t = ( vals(ID_RHOT) + hy_dens_theta_int(k) ) / r
        p = C0*(r*t)**gamma - hy_pressure_int(k)
        !Enforce vertical boundary condition and exact mass conservation
        if (k == 1 .or. k == nz+1) then
          w                = 0
          d3_vals(ID_DENS) = 0
        endif

        !Compute the flux vector with hyperviscosity
        flux(i,k,ID_DENS) = r*w     - hv_coef*d3_vals(ID_DENS)
        flux(i,k,ID_UMOM) = r*w*u   - hv_coef*d3_vals(ID_UMOM)
        flux(i,k,ID_WMOM) = r*w*w+p - hv_coef*d3_vals(ID_WMOM)
        flux(i,k,ID_RHOT) = r*w*t   - hv_coef*d3_vals(ID_RHOT)
      enddo
    enddo

!$kgen end_callsite
```

There are two ekea directives: begin_callsite and end_callsite. The code block between the two directives is the kernel block. "!$kgen" marks that the rest of the line is ekea directive. "tend_z" gives the specified code block of the kernel name.

Following Linux shell command runs Fortlab compileroption app.

```
fortlab timinggen "make miniWeather_fort.exe" --savejson miniWeather_compopts.json
```

"fortlab" is a main command to drive its subcommands. In above example, "compileroption" sub-command is used to collect compiler flags. The actual command for compilation is shown inside of double-quotation marks. The compiler flags can be collected from child processes. For example, this example uses a Makefile. You can optionally save the result to Json file using "–savejson" sub option.

Once the above command runs with success, "miniWeather_compopts.json" file will be created. The content of the json file is shown below.

```
{"etime": {
  "23": {
    "0": {
      "1": ["3.21706E+03", "3.21707E+03"],
      "2": ["3.21709E+03", "3.21710E+03"],
      "3":...
    }...
  }...
 }...
}
```

Listing 10 shows the JSON content of "timinggen" app output. Line 1 shows the type of this JSON object. In this 12 13 means the MPI rank that the timing output is generated 14 case,thetypeiselapsedtime.Thenumber"23"inline2 from. Similarly, "0" in line 3 is the OpenMP thread number. The key values in line 4-6, means the order of invocations that the kernel region is executed. The array values of each invocation are timestamps of the beginning and the end of the kernel execution.

The timing data is used to choose the best combinations of "invocation-OpenMP thread-MPI rank" that produces the kernel driving data whose timing statistics efficiently match to the statistics from the execution of the original software.

It receives a AST generated by "resolve" app. After instrumenting the code, the app compiles and runs the instrumented software to generate raw timing data, and finally it merges the timing raw data in an JSON-format file.

2.2.2 Usage

usage: fortlab-timinggen [-h] [–version] [–cleancmd CLEANCMD] [–buildcmd build command] [–runcmd run command]

[–outdir OUTDIR] [–no-cache] analysis

positional arguments:
analysis analysis object

optional arguments:

-h, --help show this help message and exit

--version show program's version number and exit

--cleancmd CLEANCMD Software clean command.

–buildcmd build command
Software build command

–runcmd run command Software run command –outdir OUTDIR output directory –no-cache force to collect timing data

This app may feed-forward following data to next app:
etimedir (type=any) elapsedtime instrumented code directory modeldir (type=any) elapsedtime data directory

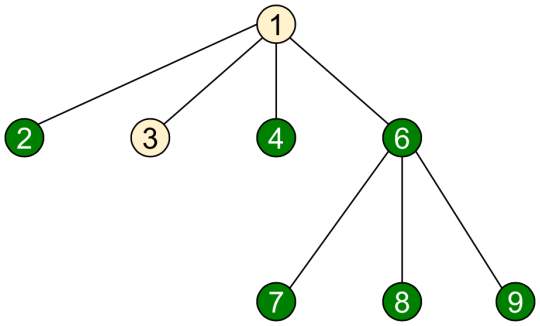
2.3 resolve app

To decide which Fortran statements should be copied in the generated kernel code, all of Fortran name references should be resolved correctly. For example, if a variable is used in a statement, the definition of the variable should be copied in the generated kernel too even the name is defined in another source file or Fortran module. This name-reference analysis is statically done using Abstract Syntax Tree("AST") generated from a Fortran parser, F2PY.

Fortran Code

```
1  : PROGRAM myprog

2  : INTEGER x = 0

3  : RPINT *, "X = ", x
4  : PRINT *, "RESULT = ", plusone(x)

5  : CONTAINS

6  : FUNCTION plusone(i) RESULT(j)
7  :     INTEGER, INTENT(IN) :: i
8  :     INTEGER :: j

9  :     j = i + 1
END FUNCTION

END PROGRAM
```

Abstract Syntax Tree (Simplified)



NOTE: Numbers in circles match to the line numbers in Fortran Code

Assuming that we want to extract "plusone(x)" at line 4 in Figure 2, the analyzer constructs AST of the entire Fortran code and decides that the dark nodes should be copied in the generated kernel. The resolution process starts with collect- ing names to be extracted. In this example of Figure 2, the resolver collects "plusone" and "x" in circle number 4. Next, the resolver moves one level up in the AST to PROGRAM statement, which is the circle number 1, and decides that circle number 2 and 6 are required. Because FUNCTION at the circle 6 includes all of circles 7,8, and 9, the three nodes under the circle 6 are also copied.

2.3.1 Example

2.3.2 Usage

usage: fortlab-resolve [-h] [–version] [–import-source srcpath] [–compile-info path] [–keep analysis]

[-i INCLUDE_INI] [-e EXCLUDE_INI] [-I INCLUDE] [-D MACRO] [–outdir OUTDIR] [–source SOURCE] [–intrinsic INTRINSIC] [–machinefile MACHINEFILE] [–debug DEBUG] [–logging LOGGING] [–invocation INVOCATION] [–data DATA] [–openmp OPENMP] [–mpi MPI] [–timing TIMING] [–prerun PRERUN] [–rebuild REBUILD] [–state-switch STATE_SWITCH] [–kernel-option KERNEL_OPTION] [–check CHECK] [–verbose VERBOSE_LEVEL] [–add-mpi-frame ADD_MPI_FRAME] [–add-cache-pollution ADD_CACHE_POLLUTION] [–repr-etime REPR_ETIME] [–repr-papi REPR_PAPI] [–repr-code REPR_CODE] path

positional arguments:

path callsite file path

optional arguments:

-h, --help	show this help message and exit
--version	show program's version number and exit
--import-source srcpath	load source file
--compile-info path	compiler flags

--keep analysis keep analysis

-i INCLUDE_INI, --include-ini INCLUDE_INI information used for analysis

-e EXCLUDE_INI, --exclude-ini EXCLUDE_INI information excluded for analysis

-I INCLUDE include path information used for analysis

-D MACRO macro information used for analysis

--outdir OUTDIR path to create outputs

--source SOURCE Setting source file related properties

--intrinsic INTRINSIC Specifying resolution for intrinsic procedures during searching

--machinefile MACHINEFILE Specifying machinefile

–debug DEBUG –logging LOGGING –invocation INVOCATION

(process, thread, invocation) pairs of kernel for data collection

--data DATA Control state data generation

--openmp OPENMP Specifying OpenMP options

--mpi MPI MPI information for data collection

--timing TIMING Timing measurement information

--prerun PRERUN prerun commands

--rebuild REBUILD rebuild controls

--state-switch STATE_SWITCH Specifying how to switch orignal sources with instrumented ones.

--kernel-option KERNEL_OPTION Specifying kernel compiler and linker options

--check CHECK Kernel correctness check information

--verbose VERBOSE_LEVEL Set the verbose level for verification output

--add-mpi-frame ADD_MPI_FRAME Add MPI frame codes in kernel_driver

--add-cache-pollution ADD_CACHE_POLLUTION Add cache pollution frame codes in kernel_driver

--repr-etime REPR_ETIME Specifying elapsedtime representativeness feature flags

--repr-papi REPR_PAPI Specifying papi counter representativeness feature flags

--repr-code REPR_CODE Specifying code coverage representativeness feature flags

This app may feed-forward following data to next app:

analysis (type=any) analysis object

2.4 kernelgen app

While the Fortran Code Analysis application in Section ref{sec:resolve} can decide which Fortran statements should be copied from the original software to the generated kernel code, we still need additional Fortran statements to make the generated kernel a stand-alone software. First, we need to provide a Fortran "PROGRAM" statement and, more importantly, statements for reading input data that drives kernel execution. This application gets the AST constructed by the "resolve" applications and manipulates the AST to be a stand-alone software. This application also converts the final AST to source files that can be compiled without errors.

This application manipulates AST in four stages in order: "created," "process," "finalize," and "flatten." At every stage, the AST is examined and changed by the application. This staged approach allows developers to synchronize modifications made on the AST so that they can ensure some modifications are available at the next stage. Once the AST manipulation is done, the final AST is converted to Fortran source files.

In many cases of kernel extraction, preparing data input for driving kernel execution is more challenging. Especially if the application uses Fortran multi-level derived types, we need to copy data in all the levels, a.k.a "deep copy." This deep-coping is automatically accomplished by this application through instrumenting the original software. The application uses Fortran WRITE statements to save scalar variables in a binary data file. In case of array type variables, the application classifies the array variables according to Fortran "type-kind-dimension" combinations and creates binary file Input/Output (I/O) subroutines per every combination of Fortran "type-kind-dimension." The generated subroutines and their call-sites are added to the original source file. In case of derived-type, the application saves binary data for all member variables as explained previously before saving the derived type itself.

The generated data is saved in a binary file. The binary file is read in the same order in the generated kernel file using subroutines that are created similarly to the Fortran "type-kind-dimension" classification. The decision on which variables should be saved in the binary file is made by analyzing reference information generated by the "resolve" application in Section ref{sec:resolve}.

It is crucial that the generated kernel produces the same output that the original software generates in the case that the user does not make any modifications in the extracted kernel. Therefore, the data generation application explained in the previous section also saves all output type variables in the binary file. This application reads the binary file and compares the content of variables between ones from kernel execution and the others from the original software. The comparison is done by calling subroutines that are created per every Fortran "type-kind-dimension" combination used in the code. If an output variable is a Fortran derived type, the application verifies the member variables in it one by one including member variables that are also Fortran derived types.

The generated kernel also has a feature of variable perturbation. Users optionally can pick an array input variable and slightly modify the value of its element at random. This feature is disabled initially and users can turn this feature on by un-commenting a perturbation subroutine call that is automatically generated in the kernel by this application. This feature is useful to measure the sensitivity of the kernel from varying input values. This application is invoked by using the "kernelgen" sub-command in either command-line or Python script. The application also optionally receives kernel timing data in JSON-format to decide when and how to save the variables in the binary file. Using the kernel timing data, this application selects subsets of the combinations of MPI ranks, OpenMP threads, and invocations of the kernel region.

2.4.1 Example

2.4.2 Usage

usage: fortlab-kernelgen [-h] [–version] [–outdir OUTDIR] [–model MODEL] [–repr-etime REPR_ETIME]
[–repr-papi REPR_PAPI] [–repr-code REPR_CODE] analysis

positional arguments:
analysis analysis object

optional arguments:

-h, --help	show this help message and exit
--version	show program's version number and exit
--outdir OUTDIR	output directory
--model MODEL	model object
--repr-etime REPR_ETIME	Specifying elapsedtime representativeness feature flags
--repr-papi REPR_PAPI	Specifying papi counter representativeness feature flags
--repr-code REPR_CODE	Specifying code coverage representativeness feature flags

This app may feed-forward following data to next app:
kerneldir (type=any) kernel generation code directory statedir (type=any) state generation code directory

2.5 vargen app

This application generates variable and function (subroutine) cross-reference information and puts the information within the source code as comment lines where the variable or function is defined as well as used. By having the information next to the variables or functions that are defined or used in the source file, users can easily navigate source codes for analysis. This cross-reference information could be useful in case it is hard to get such analysis information from a tool such as Integrated Development Environment.

The application can analyze the following information: 1) module variables used in functions, 2) caller sites for functions, 3) local variables used in the kernel block, 4) module variables used within the kernel block, and 5) code locations where module variables are referenced.

This application can be invoked as shown at the top of Figure ref{fig:ekea-varwhere}, similarly to the kernel extraction application explained in Section ref{sec:ekeaextract}. This application has almost the same operations seen in Section ref{fig:ekea-extract} except that the kernel generation part is not used. In this application, kernel source files are also created containing Fortran name cross-reference information in comment lines.

```
!Local variables possibly modified
!groupitr : derived
...
!Local variables possibly used as operand
!temperatureshortwavetendency : implicit array
...
!External variables possibly modified
!hnewinv : at module ocn_tracer_advection_mono
...
!External variables possibly used as operand
!redidiffon : at module ocn_vmix_coefs_redi
...
!!! START OF KERNEL REGION
...
!!! END OF KERNEL REGION
```

In Listing ref{lst:varwhere-output}, the comment lines contain the application-generated cross-reference information. The comment lines are written just before the kernel region where the user has specified for analysis. To reduce the amount of information, only one variable per one analysis case is shown in Listing ref{lst:varwhere-output}. The analysis information in the comment lines are only relevant to the kernel region. For example, "groupitr" variable

shown in Line 2, is a local-scope variable in the kernel region, and the comment line 1 tells that "groupitr" is possibly modified in the kernel region. In line 5, we can see that "temperatureshortwavetendency" variable is used in a Fortran statement. The analysis also applies to module variables. In line 8, the analysis tells that "hnewinv" variable may be modified during the execution of this kernel region. and Line 11 tells that "redidiffon" variable may be used in the kernel execution as an operand. To save the space, we have not shown the cross-reference analysis for function caller-callee relations.

2.5.1 Example

2.5.2 Usage

usage: fortlab-vargen [-h] [–version] [–outdir OUTDIR] analysis

positional arguments:
analysis analysis object

optional arguments:

-h, --help	show this help message and exit
--version	show program's version number and exit
--outdir OUTDIR	output directory

This app may feed-forward following data to next app:
kerneldir (type=any) kernel generation code directory

CHAPTER

THREE

BUILDING AND RUNNING FORTLAB CUSTOM APPS

While fortran has components that collectively can extract a kernel, the actual extraction process is frequently dependent on each target application. Therefore, as an interim solution, we defer the kernel extraction to fortlab-based "applications" that are customized to the target application.

Please refer to EKEA(E3SM Kernel Extraction and Analyzer: https://ekea.readthedocs.io) for more information on how a "fortlab application" is build.

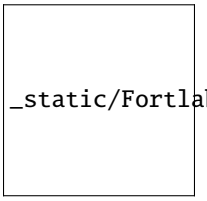
CHAPTER

FOUR

FORTLAB FRAMEWORK INTRODUCTION

FortLab is a software tool development framework on which developers can build a new kernel extraction and analysis tool for Fortran applications.

4.1 Fortlab Framework Overview



Above Figure shows a high-level view of the framework. At the core of the framework, FortLab built-in applications exist. Each built-in applications provides core features of kernel extraction and analysis capabilities. The applications can be used in a Python script as a part of another application. Or they can directly run on shell as a sub-command of "fortlab" command.

4.2 Creating a FortLab Application

The functionality of the framework is implemented as a form of FortLab application. While the application exists within the framework, it acts as if it is an independent application as explained in the following section.

Following code shows a simple example of a FortLab application that reads a name as a command-line argument and greets with the name.

```
1   class Hello(App):
2     "greet a name"
3     _name_ = "hello"
4     _version_ = "0.1.0"
5
6     def __init__(self, mgr):
7       self.add_argument("name", help="input name")
8
9     def perform(self, args):
10      print("Hello "+args.name["_"])
```

At line 1, "App" class is imported from the "fortlab" Python module. At line 2 a new class "Hello" is created from the "App" class. Line 3 has a short description of the application. "_name_" in line 4 sets the name of this application.

The name is used as a sub-command of "fortlab" in command-line. "_version_" at line 5 sets the version of this implementation. "__init__" function at line 6 is optional. Within the function, user can define the command-line argument as shown in line 7-8. The "add_argument" function is almost similar to the interface of Python's "argparse" standard module. "perform" function at line 9 is the entry of the application's execution and it is mandatory. The "args" arguments of the function is the way to receive user's input at command-line. There is no special restriction on what to add in the body of the "perform" function. In this case, a greeting string is printed with the input name. Line 10 shows how to access the user's input through the "args" variable. The square bracket notation is used to indicate the argument type conversion method. The underline string in the brackets indicates "No conversion", and therefore, "args.name" is a type of string.

4.3 Running a FortLab Application in command-line

Following bash commands show how to run the application ("app") that we created in the above code and the output from the run. In line 1, "fortlab" is used as a shell command. The double dashes separate the command from the sub-command. Next, a path to a file that contains the app follows. The framework automatically detects the app in the Python module-level objects. If there exists more than one apps in the module, a hash mark "#" with a class name can be used to specify a particular app class in the module. The output of the run shows in line 2. Line 3 to the end demonstrates the framework's capabilities that show version and usage of the app on screen. Note that the version and the short description in the Hello class are shown in the screen output, which are generated by the framework.

```
1  >> fortlab -- hello.py world
2  Hello world
3  >> fortlab -- hello.py --version
4  hello 0.1.0
5  >> fortlab -- hello.py --help
6  usage: fortlab-hello [-h] [--version]
7                       name
8  greet a name
9
10 positional arguments:
11   name         input name
12
13 optional arguments:
14   -h, --help   show this help message
15   --version    show program's version
```

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search