Documentation of the FEAST & Featflow2 build system

Sven H.M. Buijssen

October 22, 2010

Contents

1 Preface

That's how it is with people -
nobody cares how it works as long as it works.
Councillor Hamann, Matrix Reloaded

This document describes the configure and build systems developed for FEAST – and ported to FEATFLOW2 recently – by explaining in detail how `configure`, `f90cpp` `Makefile.inc`, `Makefile.buildID.inc`, `Makefile.cpu.inc`, some less prominent scripts and GNU `make` interact. The whole infrastructure has been set up to provide a convenient and almost fully automated way to port and compile a FEAST application on numerous differing platforms, ranging from small Linux installations on a laptop or personal computer over medium-sized commodity based clusters up to vector machines like NEC SX-8. FEATFLOW2 applications can now use the same infrastructure. If everything works as intended, the user merely has to invoke two commands to compile his application: `configure` and `make`. In the rare cases where this procedure returns an error message, usually only a few changes have to be made to the compile system, which are discussed in section 6.2, but particularly in section 7 on page 41.

The document starts with a quick start guide on using `configure` to create a `GNUmakefile` for a FEATFLOW2 application. Then, the purpose of Makefiles in general is explained. The next chapter introduces the concept of build IDs, a string characterising architecture, cpu, operating system, compiler, BLAS library and possibly MPI environment. These build IDs are used to define appropriate compiler names and compiler settings. Subsequently, all the command line options and implementation details concerning `configure` are presented. Chapter 6 explains the most prominent targets in `configure`-generated Makefiles and is recommended reading for everyone. It is followed by a chapter explaining how the Makefile expands build IDs to compiler names and compiler settings. The next chapters are dedicated to the FEAST preprocessor, a script that performs a lot of magic unnoticeably, is crucial to keep FEAST (and in future most likely FEATFLOW2) portable and is already actively exploited by the FEATFLOW2 application `flagship`, too. Finally, it is explained how third party libraries like BLAS , LAPACK, UMFPACK etc. are handled as well as how one can swap one BLAS library for another, e. g. to use the highly tuned Intel Math Kernel Library (MKL) instead of a self-compiled BLAS and LAPACK library. The last chapter points the user to the relevant sections of this document in case th build system throws an error message.

Having worked his way through all the chapters, the user will understand how scripts, interpreters, preprocessors and compilers work together to compile a FEAST or FEATFLOW2 application on a desktop computer or workstation and how to run it.

2 Quick Start Guide

You can use the established FEATFLOW2 build system along with the `configure`-based build system that has been developed for FEAST and ported recently to FEATFLOW2. The established FEATFLOW2 build system uses static Makefiles called `Makefile`, the `configure`-based build system creates Makefiles called `GNUmakefile`. Be aware that when using GNU make, the latter Makefile will take precedence of the former. Be sure to first remove the file `GNUmakefile`, if you do not want to use the `configure`-based build system any longer and want to return to the established FEATFLOW2 build system.

Let us assume your application is stored under `Featflow2/area51/bouss2dmini` and you want to try out the new build system. Source files of your application are organised in a subdirectory called `src` and are as follows

```
bouss2dmini.f90  bouss2dmini_callback.f90  bouss2dmini_mit_1.f90  bouss2dmini_mit_2.f90
```

The source file providing the main program is `bouss2dmini.f90`. Let us further assume that you want to name the binary `bouss`. Then you would issue the following command (line breaks are introduced only for readability):

```
../../bin/configure \
      --appname=bouss \
      --programfile=src/bouss2dmini.f90 \
      --srclist_app="src/bouss2dmini_callback.f90 src/bouss2dmini_mit_1.f90 \
                     src/bouss2dmini_mit_2.f90"
```

Note that adding the program file to the list of source files is optional. There is a shortcut for specifying a complete list of source files. Use the backtick operator (`) and issue an `ls` command on the `src` subdirectory to get a list of all Fortran 90 source files in there and use this as argument to `--srclist_app`:

```
../../bin/configure \
      --appname=bouss \
      --programfile=src/bouss2dmini.f90 \
      --srclist_app="`\ls src/*.f90`"
```

It is probably convenient to store these instructions in a small shell script in your application directory. Most users prefer to name this script consistently `configure`, too.

Even if you happen to not be a tidy programmer – i. e. old copies of your code usually linger around in the `src` subdirectory, e.g. `src/bouss2dmini.old.f90, src/bouss2d-unpatched.f90, src/broken.f90` – you will *not* run into a problem when using this backtick operator trick. Surely, there will be a build rule in the resulting `Makefile` for an object file corresponding to any of these obsolete source files. But these object files do not become prerequisites for the binary and hence will not be compiled at all when you issue `make`! This is because the prerequisite list of the binary is *not built* by concatenating all FEATFLOW2 kernel source files plus the source files specified via the `--srclist_app` option. `configure` is sophisticated enough to actually parse the source files to set up a minimal list of source files that have to be compiled in order to be able to link the binary. It does so by parsing first the program source file – `src/bouss2dmini.f90` in the example above – and looking for Fortran 90-style `USE` and `INCLUDE` statements (case-insensitively) as well as C-style `#include` statements. Dependencies are then recursively parsed for their dependencies. This algorithm ensures that unused or obsolete source files (e. g. copies of old source code) will not pose a problem.

Upon completion `configure` prints a short list of the settings applied and creates a Makefile named `GNUmakefile`. Invoke

```
make
```

to compile your application. Issue

```
make help
```

and

```
../../bin/configure --help
```

for a list of options. Please see the next chapters for a detailed description of the Feast build system made available to Featflow2 in the beginning of March 2009.

3 `make` and Makefiles

Before talking about how to let FEAST's `configure` create a `Makefile` for a FEAST application it might be worth to say a word about `make` and Makefiles in general. Users familiar with this general concept in software development may skip the paragraph and proceed to the next. `make` is a (command line) utility for automatically building large applications. Generally speaking, it is an expert system that, based on rules defined in an input file called `Makefile`[1], tracks which files have changed since the last time the project was built and invokes the compiler(s) only on those source code files and their dependants, in the correct order. With the rise of Integrated Development Environments (IDE) it becomes less common to use a `Makefile` explicitely as the build process is dealt with silently behind the scenes, usually with similar expert systems.[2] But especially in Unix-based platforms `make` remains widely used.

Writing and maintaining Makefiles manually is cumbersome and error-prone. In fact, it is hard to keep track of all dependencies between source files; besides that writing Makefiles is not harder than writing shell scripts. Neglecting a dependency or forgetting to add a new one along with a new revision of a source file might not have as drastic results as rendering compilation impossible (e. g. a compiler complaining about a missing dependency or a link error). The impact usually is more subtle: A source file whose dependency list was not been updated, does not get recompiled if the dependency is updated. Interface changes will go undetected. In case of optimised builds, inexplicable runtime crashes may occur because of wrong multifile interprocedural optimisations. Calculation results might be wrong because some routines do not get executed as expected. Enough reasons why in general dependency lists for Makefiles are created automatically.[3]

Makefiles not only describe dependencies between source files, they usually also contain platform specific build instructions. Compilers differ among each other, not only between different vendors, but also between different releases of the same compiler. Command line options for one compiler release might be regarded as deprecated by the same compiler released only a few months later. The opposite holds true in particular for command line options that control optimisation levels. Older compilers may not know about an option a more recent compiler accepts and will refuse to compile the code. It is unrealistic to believe that this variety of options can be reliably remembered. It is much more convenient to let a `Makefile` sort things out and come up with appropriate settings for the compiler release detected at invocation time.

Besides incompatible command line options there is also the issue of portability. Depending on the vendor's interpretation of the language standard some code sections need to be custom tailored to a particular compiler (or compiler release). The same holds true for the maturity level of compilers: Compilers are nothing but pieces of software and as such not free of flaws. A big software package like FEAST is likely to stumble upon a number of compiler bugs[4]. Usually a workaround can be found such that drastic measures like revocation of FEAST support for this particular compiler are not necessary. But being a workaround for a particular compiler, most probably merely for a particular compiler release, there is no point in enabling this workaround for all compilers. Code should not be designed to not raise compiler bugs, but instead should be clearly

[1] GNU Make actually tries the following names, in order: `GNUmakefile`, `makefile` and `Makefile`.

[2] At the expense of reduced flexibility, though.

[3] By means of e. g. `sfmakedepend`, `mkdeps`, `cpp -M` etc. A reasonably new player in these field is `TCBuild` (`http://www.macresearch.org/tcbuild-new-build-tool-fortran`), a Python script released in February 2008 that aims to do a similar job as FEAST's `configure` does.

[4] It is regarded a compiler bug if valid code causes a compile error. It is likewise called if valid code is miscompiled leading to wrong results or, not necessarily considered worse, runtime crashes. A prominent example in FEAST is the incapability of Intel Fortran Compiler releases 8.x and 9.x to perform Fortran 90 array operations with vectors of length greater than one million. Attempts to do so crash the code. (To be more precise, they lead to a runtime crash depending on the stack size settings: if the operation fits onto the stack, there is no problem. But not on all systems a non-privileged user is allowed to set the stack size to unlimited.)

comprehensible. A workaround should be perspicuously identifiable as such and only active for the compiler (or compiler release) it was written for. This is where the preprocessor enters the scene. By its means it is possible to present every compiler custom tailored source code. Code blocks that trigger a particular undesired compiler behaviour (internal compiler errors, wrong code generation, unorthodox interpretation of the language standard) are replaced by a suitable alternative, encapsulated by a descriptive preprocessor macro. Compilers not affected by the same compiler bug are still presented with the original, probably clearer code variant instead of the workaround version. A developer reading such code fragments sees the preprocessor directives and immediately knows why something is implemented in the way it is. Without the possibility of providing multiple alternatives by means of a preprocessor, a developer, in particular someone not familiar with the history of the workaround, would be tempted to implement it "in a more easy, clearer way". The alternate code can be quite easily removed again if an updated compiler release fixes the bug, simply by searching for preprocessor directives referencing the compiler. Which of the preprocessor directives is active at a time is controlled by command line options passed along with the compile command. To learn more about how this is portably implemented in FEAST, see section 8 on page 44.

As with optimisation options remembering the appropriate preprocessor command line options for every compiler (release) is next to impossible. Again, Makefiles are set up to take care of these.

The use of Makefiles is not restricted to compilation of programs, though this is what they are written for in the overwhelming number of cases. But there is nothing that deters from coding a `Makefile` to e. g. typeset a LaTeX document. FEAST's documentation is written in LaTeX. So is, amongst other documents, the present document. Makefiles control the process of typesetting. A document is only recompiled if the master file, included LaTeX source files, a bibliography file or an embedded graphic have changed.

3.1 Technical details on Feast Makefiles

`make` ships with every Single UNIX® Specification-compliant system. But there exist a number of different implementations, each with their own non-standard enhancements. FEAST's Makefiles require GNU Make (version 3.80 or higher). The reason for this is that they rely on two features beyond the reference implementation; together these features are only available in recent versions of GNU Make: order-only prerequisites and conditionals.

Order-only prerequisites are needed in FEAST Makefiles for the following reason: To avoid cluttering up a working directory, FEAST Makefiles allow for storing object files in separate directories. They can even be stored in a directory completely unrelated to the working directory. This feature is particularly useful on systems where disk quotas are enforced. When disk space is limited, source files should remain in the user's home directory (in order to have them automatically backed up on a regular basis). Object files can be stored in some kind of scratch area.[5] The rules for object files rely on the object directory to exist. Naively arguing, it would be sufficient to have a rule that creates a directory if it does not exist. But the time stamp of a directory is updated every time a file system object is added to or removed from it. It is obvious that every source file would be recompiled every time `make` is invoked if the object directory were a normal prerequisite, as there is always a first and a last source file to be compiled. Compiling the latter updates the time stamp of the object directory such that this time stamp is newer than that of the object file derived from compiling the first source file.

A second feature present in GNU Make version 3.80 or higher – and not required by the Single UNIX® Specification – are conditionals. Citing from the GNU Make documentation: "A conditional causes part of a `Makefile` to be obeyed or ignored depending on the values of variables. Conditionals can compare the value of one variable to another, or the value of a variable to a constant string. Conditionals control what `make` actually 'sees' in the `Makefile`."[6] The flexibility of FEAST's Makefiles with respect to supporting differing platforms, compilers, compiler releases, BLAS implementations (and MPI environments) and automatic

[5] See configure options `--objdir-prefix` and `objdir-lib-prefix`, page 23.
[6] See `http://www.gnu.org/software/make/manual/`.

detection of appropriate default settings for a given environment relies upon conditionals. The mechanisms are explained in detail in section 7 on page 41.

4 Build ID

A term omnipresent in the next chapters is “build ID”. It identifies a given set of settings to be used for compilation. The concept of build IDs is explained in the following.

Due to ongoing joint efforts FEAST is reasonably portable. It has been successfully compiled and run on numerous differing platforms, on small Linux boxes, commodity based clusters, and – being a HPC FEM package – traditional supercomputers. Being portable does not mean that there is merely one known set of conditions under which FEAST can be compiled on a particular platform. “FEAST is portable” comprises being able to compile FEAST on the same hardware with a wide range of compilers, to be able to use multiple MPI environments, to be able to use an arbitrary BLAS implementation (usually combined with a LAPACK implementation into one single library, e. g. ACML, MKL, to some extend GOTO BLAS) etc. To easily exploit the flexibility provided by FEAST a given hardware and software environment is characterised and identified by a string called “build ID” consisting of five or six tokens. The sixth token is only necessary in case the binary is intended to be compiled for parallel execution:

1. architecture
2. cpu
3. operating system
4. compiler
5. BLAS implementation
6. (MPI environment)

The tokens are separated by a dash. Examples for valid build IDs are:

```
alpha-ev6-osf1-cf90-dxml,
alpha-ev6-osf1-cf90-dxml-dmpi,
ibm-powerpc_power4-aix-xlf-essl-poempi,
pc-coreduo-linux64-intel-goto-ompi,
pc-opteron-linux32-g95-goto,
pc-opteron-linux32-g95-goto-lammpi,
pc-pentium4m-linux32-pgi-atlas,
pc-pentium4m-linux32-pgi-atlas-ompi,
sun4u-sparcv8-sunos-g95-blas-mpich2,
sun4u-sparcv9-sunos-sunstudio-perf-lammpi,
sx8-none-superux-f90-keisan-mpi.
```

Under the hood, the build ID is expanded, e. g. by adding information about serial or parallel execution (a serial build does not need any MPI header files nor libraries), and the availability of co-processors (like GPUs or Cell). Section 7 on page 41 will deal with this in full detail.

5 configure

This chapter requires understanding of the terms `Makefile` and build ID as explained in previous chapters.

FEAST's – and now FEATFLOW2's – `configure` is a Perl script which basically does the following: It determines a set of values necessary to compile a FEAST application to be run on a given machine using a particular environment (MPI, BLAS, LAPACK, UMFPACK etc.), it then finds the dependencies between the source files and finally creates a `Makefile` for the application. Anyone familiar with compiling software from source files and particularly users familiar with compiling open source software will recognise the name: `configure`. *But FEAST/FEATFLOW2's configure is not GNU configure!* Why? There are three reasons: First, FEAST and FEATFLOW2are developed by mathematicians and computer scientists, not lawyers. The implications of using a piece of software released under the GNU General Public License, i. e. GNU `autoconf` and GNU `automake` in order to produce a GNU configure script, for the license model under which FEAST and FEATFLOW2 are eventually released are unclear to us. The intention is to do mathematics and scientific computation, not to dive into legal mazes. Second, GNU configure was designed having in mind mainly C and C++ programs. The diversity of C and C++ implementations is far greater than it is the case in the Fortran world. To compile Fortran 90/95 code it is not necessary to first check a dozen possible locations to find the definition of a particular symbol. Finally, it was simply easier to write a Perl script to do the job (and to gradually extend its capabilities) than to learn the GNU autoconf syntax. Whenever there is some new feature required, it can be added quite easily (as long as one speaks Perl reasonably well). The fact that maintaining FEAST/FEATFLOW2's `configure` obviously cannot benefit from the work of the open source community is balanced by not having to cope with the associated set of nightmares and other dependencies (due to unnecessary flexibility) that GNU `configure` implies. The script can easily be expanded to support new developments in FEAST/FEATFLOW2, like GPU or Cell support. It is unclear whether it would have been that easy with GNU `configure`.

The purpose of FEAST/FEATFLOW2's `configure` is to create a `Makefile` for a FEAST/FEATFLOW2 application[1] – and to perform all the necessary tests to be able to successfully generate one. Such tests include whether all source files exist (especially the stated application-specific ones), whether any symbolic link needs restoration, whether all external software prerequisites such as a matching version of GNU Make are met and whether any given build ID is valid. Any additional test is left to the time when the generated `Makefile` is executed by `make` because once a `Makefile` has been created, it can easily be edited (and broken) by the user prior to executing it. Makefile variables may also be overridden on the command line. Even compiler settings might have changed since the `Makefile` was created. All these possibilities require that at `make` runtime checks are performed whether all compilers are available, whether the requested BLAS version is in the search path of the compiler and linker, whether Makefile variables are valid[2] and do not collide[3], to name just a few. `configure`-generated Makefiles even check whether the compiler or the compiler settings have changed since the last time `make` has been invoked to prevent intricate linker or runtime problems (see section 6.2.1 on page 34 for more information on this feature).

FEAST/FEATFLOW2's `configure` accepts a great deal of command line arguments to override its defaults. A complete, up-to-date list of command line options along with a short description of every option is always available via

[1]The only exception to this rule is related to the configure option `--regression-benchmark` which creates one `Makefile` plus a `Makefile` in every subdirectory with a `src_` prefix.

[2]Example: The optimisation level is controlled via the Makefile variable `OPT` and it may take exactly one of three values: `NO`, `YES` or `EXPENSIVE`. Any other value raises an error.

[3]Example: The Makefile variable `MODE=PARALLEL`, used to indicate that a FEAST/FEATFLOW2 application should be compiled for parallel execution, must not be used in combination with the preprocessor directive `-DENABLE_SERIAL_BUILD`. Attempts to mix them should not go undetected.

```
./configure --help
```

`configure` provides reasonable defaults settings for all options, but in general can not possibly know the application specific source files. Directly calling `configure` in an application directory

```
../../bin/configure
```

will most likely not result in a usuable `Makefile`. Most users will need to customise a few values: e. g. application name, name of main program source file, list of application-specific source files. These users are advised to write a small shell script, a wrapper providing default values for the most frequently used command line arguments that passes all remaining command line arguments to the real `configure` script tranparently. The name of this script obviously does not matter, but for consistency reasons it is usually named `configure` as well. An example wrapper as could be used in a FEATFLOW2 checkout in `applications/cc2d` is

```
#!/bin/sh

../../bin/configure \
      --appname=cc2d \
      --programfile=src/cc2d.f90 \
      --srclist_app="`ls src/*.f90`" \
      \$@
```

A more complex example that stored object files to a file space that is not backup'd (a procedure that allows it to develop in one's quota-restricted home directory as source files take up less space than a bunch of object files for different build IDs) can be found in any FEAST checkout in `applications/tutorial`:

```
#!/bin/sh
# This shell script invokes the global FEAST configure script
# which in turn will determine all dependencies of the current
# application and create a Makefile for it.
# For valid command line options to this global FEAST configure
# script, invoke it with '--help'.

######################################################
# Don't let the script be confused by non-english messages
# from system information programs.
# (LC_ALL overrides the value of the LANG environment variable
# and the values of any other LC_* environment variables.)
LC_ALL=C


######################################################
# Change to the directory where this script resides
# (you might call this script via <some path>/configure,
# so change to <some path> first.
SCRIPTPATH=`dirname $0`
cd ${SCRIPTPATH}


######################################################
# Default behaviour:
# Do not store object files to home directory, but to some scratch area
# without quota restrictions. Try to come up with a reasonable name
# for the object file directories (per application and one for the libraries
# which are shared among all FEAST applications).
#
# Q: Why not store object files directly to the home directory, to the
# same directory where FEAST is installed?
# A: At the Faculty of Mathematics, TU Dortmund and the compute servers
# LiDO, JUMP and NEC file system disk quotas are enforced. But everywhere
# a scratch area providing vast disk space is available - usually without
# backup. But backup is not needed for object files.
#
# Note: this default behaviour can always be overridden by explicitly
# setting --objdir=<some path> on the command line when invoking this
# script!
#
# Specify its directory prefix here.
BASEDIR_OBJFILES=${HOME}/nobackup/feastobj

# To support multiple working copies of FEAST which all store their
# object files beneath ${BASEDIR_OBJFILES}, but which should not interfere
# which each other, duplicate the hierarchy of these FEAST installations
# beneath ${BASEDIR_OBJFILES}. The procedure can be explained most easily
# with an example:
# Consider you have two FEAST installations, one in $HOME/feast and another
# in $HOME/tmp/feast, then the object files should go to ${BASEDIR_OBJFILES}/feast
# and ${BASEDIR_OBJFILES}/tmp/feast, respectively.
CWD=‘/bin/pwd‘

# Try to shorten the directory hierarchy below ${BASEDIR_OBJFILES}.
# Why? Because it is used to create the object directory and some
# compilers enforce (silently) restrictions on include and module
# directories (e.g. PGI, see below). Do not let path to object
# directory become too long!

# Step 1:
#   try to cut off the leading part ending in your username.
FEASTINSTDIR=${CWD}
# $USER is used later on, but not all Unix systems define
# this environment variable, Sun Solaris e.g. does not.
test -z "$USER" && USER="$LOGNAME"
# Username contained in path name?
FEASTINSTDIR=‘echo ${FEASTINSTDIR} | sed "s/^.*\/$USER\//\//;"‘;

# Step 2:
#   remove "feast/feast/"
FEASTINSTDIR=‘echo ${FEASTINSTDIR} | sed "s/\/feast\/feast//;"‘;

# Concatenate directory strings to come up with a directory name
# for FEAST application object files
OBJDIRPREFIX=${BASEDIR_OBJFILES}${FEASTINSTDIR}

# Create a likewise directory for all libraries which should
# be shared among all FEAST applications as - being a library -
# they have no dependency on the FEAST kernel or a FEAST application
# and are all compiled with identical settings.
OBJDIRLIBPREFIX=‘dirname \‘dirname ${OBJDIRPREFIX}\‘‘


#####################################################
# Sanity check. Don’t let path to object directory become too long!
# (The PGI compiler cannot handle object directories exceeding a length
#  of 120 characters. Using it to store files works, using it to look
#  for module files works, using it as include path (for header files)
#  does *not* work - *without* any error message! Header files are simply
#  reported to not be found.)
if [ "`echo ${OBJDIRPREFIX} | wc -c`" -gt "120" ]; then
    echo "$0: ERROR."
    echo "    Sorry, the object directory path exceeds 120 characters."
    echo "    This may lead to problems with certain compilers (e.g. PGI"
    echo "    will not find header files in this directory)."
    echo "    Please abbreviate the object directory which is currently"
    echo "    set to:"
    echo "<"${OBJDIRPREFIX}">"
    echo
    echo "Application not configured."
    exit 1
fi


#####################################################
# Finally, invoke the real work horse, FEAST's configure, and pass all
# configuration settings determined above to it. Command-line parameters
# given when invoking this script are passed to the 'real' configure
# script as well (via "$@"). They can be used to override settings
# hard coded here.
../../bin/configure \
    --no-opt \
    --appname="feastutor" \
    --srclist_app="slavemod.f90 userdef.f90" \
    --apponlyflags=" -DENABLE_LOG_STYLE_CATEGORYTAGS -DENABLE_LOG_ADDMSGLEVEL \
                   -DENABLE_LOG_ADDMASTERSLAVE -DENABLE_LOG_ADDTIMESTAMP" \
    --objdir-prefix=${OBJDIRPREFIX} \
    --objdir-lib-prefix=${OBJDIRLIBPREFIX} \
    "\$@"
```

A `configure` wrapper file in an application directory should not be confused with the central Perl script `configure` in `bin`, though.

In the following the options will be introduced in alphabetical order and explained in full detail.

It might be worth mentioning, though, that `configure` does not store compiler and compiler settings. These are contained in plain Makefile syntax in `Featflow2/Makefile.cpu.inc` and the Makefile templates, i. e. files that match `Featflow2/templates/*.mk`.

`--appname=<file>`

Specifies the name of the executable file.

This option sets the Makefile variable `APPNAME` to `<file>`. The default is to name the application `feat2app`[4].

`--apponlyflags=<string>`

Specifies preprocessor and compiler command line options for the application.

This option sets the Makefile variable `APPONLYFLAGS` to `<string>`. The additional options are not used to compile libraries, they are only applied for compilation of kernel and application source files. Both FEAST and FEATFLOW2 store library object files and libraries themselves in an object directory shared among all applications and stores all kernel and application's object files in a separate object directory per application. The benefit of this option is that libraries can still be shared despite the fact that the command line options for FEAST/FEATFLOW2 applications differ. Typically, the option is used to pass along application-specific preprocessor flags.

[4]The flagship application overrides this setting to `flagship`.

By default, the value of the Makefile variable `APPONLYFLAGS` is empty.[5]

`--ar=<file>`

Specifies the name of the ar command used to create and modify archives. The option overrides any defaults automatically determined based on the chosen build ID.

This option sets the Makefile variable `AR` to `<file>`. The default is to not prescribe a value as the setting is chosen in the appropriate Makefile template for the chosen operating system.

`--buildlib=ARG`

Specifies the names of the libraries to be built along with the Feast or Featflow2 application. The option overrides any defaults automatically determined based on the chosen build ID.

Items in `<string>` may either be separated by whitespace or commas. The option can be used multiple times on the same command line, the respective `<string>` arguments are concatenated. The following examples are equivalent:

```
./configure --buildlib="metis amd umfpack4 lapack"
./configure --buildlib=metis,amd,umfpack4 lapack
./configure --buildlib="metis amd" --buildlib="umfpack4,lapack"
```

This option sets the Makefile variable `BUILDLIB` to `<string>`. The default is to not prescribe a value as the appropriate settings are chosen – depending on the BLAS, LAPACK, METIS and UMFPACK implementations – in `Makefile.cpu.inc` and several of the Makefile templates.

`--cc=<file>`

Specifies the name of the C compiler overriding any defaults automatically determined based on the chosen build ID.

This option sets the Makefile variable `CC` to `<file>`. The default is to not prescribe a value as the setting is chosen in the appropriate Makefile template for the chosen compiler.

`--cflagsc=<string>`

Specifies command line options for the C compiler overriding any defaults automatically determined based on the chosen build ID.

Items in `<string>` may either be separated by whitespace or commas. The option can be used multiple times on the same command line, the respective `<string>` arguments are concatenated. The following examples are equivalent:

```
./configure --cflagsc="-xW -DUSE_COMPILER_INTEL -O3"
./configure --cflagsc=-xW,-DUSE_COMPILER_INTEL,-O3
./configure --cflagsc="-xW" --cflagsc="-DUSE_COMPILER_INTEL" --cflagsc="-O3"
```

This option sets the Makefile variable `CFLAGSC` to `<string>`. The default is to not prescribe a value as the appropriate settings are chosen – depending on the optimisation level, compiler and BLAS implementation and possibly the MPI environment – in `Makefile.cpu.inc` and several of the Makefile templates.

`--cflagscxx=<string>`

Specifies command line options for the C++ compiler overriding any defaults automatically determined based on the chosen build ID.

Items in `<string>` may either be separated by whitespace or commas. The option can be used multiple times on the same command line, the respective `<string>` arguments are concatenated. See option `--cflagsc` for example usage.

This option sets the Makefile variable `CFLAGSCXX` to `<string>`. The default is to not prescribe a value as the appropriate settings are chosen – depending on the optimisation level, compiler and BLAS implementation and possibly the MPI environment – in `Makefile.cpu.inc` and several of the Makefile templates.

[5] The flagship application again illustrates the usage of this parameter as it uses it to pass the current date into the binary.

`--cflagsf77=<string>`

Specifies command line options for the Fortran 77 compiler overriding any defaults automatically determined based on the chosen build ID.

Items in `<string>` may either be separated by whitespace or commas. The option can be used multiple times on the same command line, the respective `<string>` arguments are concatenated. See option `--cflagsc` for example usage.

This option sets the Makefile variable `CFLAGSF77` to `<string>`. The default is to not prescribe a value as the appropriate settings are chosen – depending on the optimisation level, compiler and BLAS implementation and possibly the MPI environment – in `Makefile.cpu.inc` and several of the Makefile templates.

`--cflagsf90=<string>`

Specifies command line options for the Fortran 90 compiler overriding any defaults automatically determined based on the chosen build ID.

Items in `<string>` may either be separated by whitespace or commas. The option can be used multiple times on the same command line, the respective `<string>` arguments are concatenated. See option `--cflagsc` for example usage.

This option sets the Makefile variable `CFLAGSF90` to `<string>`. The default is to not prescribe a value as the appropriate settings are chosen – depending on the optimisation level, compiler and BLAS implementation and possibly the MPI environment – in `Makefile.cpu.inc` and several of the Makefile templates.

`--coprocessor=<string>`

Specifies that the application should be compiled to use hardware accelerator `<string>`. Currently supported are the NVIDIA Geforce FX (`geforcefx`) and Geforce 6, 7 and 8 GPUs (`geforce6`).

This option adds the Makefile variable `COPROCESSOR` and sets it to `<string>`. The default is to not use a hardware accelerator in Feast, a `Makefile` is generated that is completely bare of instructions to use hardware coprocessors.[6]

`--cpp=<file>`

Specifies the name of the C preprocessor overriding any defaults automatically determined based on the chosen build ID.

This option sets the Makefile variable `CPP` to `<file>`. The default is to not prescribe a value as the appropriate setting is chosen in the appropriate Makefile template for the platform.

`--cxx=<file>`

Specifies the name of the C++ compiler overriding any defaults automatically determined based on the chosen build ID.

This option sets the Makefile variable `CXX` to `<file>`. The default is to not prescribe a value as the appropriate setting is chosen in the appropriate Makefile template for the chosen compiler.

`--debug` Requests that a list is printed to the screen while generating the Makefile comprising all Makefile variables that are explicitly overridden by configure command line options. Their values will take precedence in the generated Makefile over the default values set in `Featflow2/Makefile.cpu.inc` and the template files in `Featflow2/templates/*.mk` for these variables.

The option has no influence on the Makefile that is generated. Particularly, it is not the same as specifying `--no-opt`.

[6]This means it is not possible to dynamically request coprocessor usage on the `make` command line via `make COPROCESSOR=geforcefx`. Any setting of `COPROCESSOR` is ignored. This fact makes the option "special" in the sense that it is the only case where a configure option can not be overridden again on the `make` command line. The application needs to be re-configured with this option to generate a new Makefile that contains instructions on how to compile Feast with coprocessor support.

`--f77=<file>`
Specifies the name of the Fortran 77 compiler overriding any defaults automatically determined based on the chosen build ID.

This option sets the Makefile variable `F77` to `<file>`. The default is to not prescribe a value as the setting is chosen in the appropriate compiler Makefile template.

`--f90=<file>`
Specifies the name of the Fortran 90 compiler overriding any defaults automatically determined based on the chosen build ID.

This option sets the Makefile variable `F90` to `<file>`. The default is to not prescribe a value as the setting is chosen in the appropriate compiler Makefile template.

`--force-id`
Forces a particular build ID to be used.

A FEAST `Makefile` checks by default whether a build ID matches the current host, i. e. whether the first three tokens of the build ID match the value returned by `bin/guess_id`. This option sets the Makefile variable `FORCE_ID` to `NO` which disables the sanity check. When cross-compiling FEAST, e. g. on the gateway host of NEC SX-8, the user needs to specify this option along with an appropriate setting of `--id`.

`--gpuflags=<string>`
Specifies preprocessor and compiler command line options for GPU hardware accelerators. Typically, this option is used in combination with `--coprocessor`.

This option redefines the Makefile variables `CFLAGSF90` and `CFLAGSCXX` by appending all substrings, given as whitespace- or comma-separated list, in `<string>`. The option is similar to `--apponlyflags`, but in contrast to it affects `CFLAGSCXX`, too.

For documentation of valid arguments see `feastcppswitches.h` in FEAST and `feat2cppswitches.h` in FEATFLOW2.

`--header-files-to-copy=<string>`
Specifies a list of the header files that must not be evaluated by the C preprocessor`CPP`, but directly transferred to the object directory.

Items in `<string>` may either be separated by whitespace or commas. The option can be used multiple times on the same command line, the respective `<string>` arguments are concatenated.[7]

Typically, this option is required for header files that contain C preprocessor directives (e.g. `#define`) that influence compilation in depending header and source files or that dependent themselves on settings present in other header files. Any path information present in C- (`#include`) and Fortran-style include statements (`include`) is automatically removed when transferred to the object directory.

`--help` Prints a short version of this overview of available command line options. No Makefile is created, no existing Makefile is touched.

`--id=<build_id>`
Prescribes the build ID `<build_id>` to be used instead of leaving the choice to `Makefile.cpu.inc` where defaults are defined for any platform FEAST has been ported to.

For a list of valid build IDs for the current host invoke

```
./configure --list-ids
```

For a complete list of valid build IDs invoke

```
./configure --list-all-ids
```

[7]The flagship application again illustrates the usage of this parameter as it uses it for two header files.

`--inc=<string>`

Specifies the string that is passed to the compiler in order to add particular directories to the compiler's include path (used to search for module files (`USE` statement) and include files (`INCLUDE` statement)). The option overrides any defaults automatically determined based on the chosen build ID.

Items in `<string>` may either be separated by whitespace or commas. The option can be used multiple times on the same command line, the respective `<string>` arguments are concatenated. The following examples are equivalent:

```
./configure --inc="-I/usr/local/include -I/home/feastuser/include"
./configure --inc=-I/usr/local/include --inc=-I/home/feastuser/include
```

This option sets the Makefile variable `INC` to `<string>`. The default is to not prescribe a value as the appropriate settings are chosen – depending on the BLAS, LAPACK, METIS and UMFPACK implementations – in `Makefile.cpu.inc` and several of the Makefile templates.

`--ld=<file>`

Specifies the name of the linker overriding any defaults automatically determined based on the chosen build ID.

This option sets the Makefile variable `LD` to `<file>`. The default is to not prescribe a value as the setting is chosen in the appropriate compiler Makefile template.

`--ldflags=<string>`

Specifies command line options for the linker overriding any defaults automatically determined based on the chosen build ID.

Items in `<string>` may either be separated by whitespace or commas. The option can be used multiple times on the same command line, the respective `<string>` arguments are concatenated. See option `--cflagsc` for example usage.

This option sets the Makefile variable `LDFLAGS` to `<string>`. The default is to not prescribe a value as the appropriate settings are chosen – depending on the optimisation level, compiler and BLAS implementation and possibly the MPI environment – in `Makefile.cpu.inc` and several of the Makefile templates.

`--libdir=<string>`

Specifies the `<string>` that is passed to the linker in order to have the linker search in particular directories before searching the standard directories. The option overrides any defaults automatically determined based on the chosen build ID.

Items in `<string>` may either be separated by whitespace or commas. The option can be used multiple times on the same command line, the respective `<string>` arguments are concatenated. The following examples are equivalent:

```
./configure --libdir="-L/usr/local/openmpi -L/usr/local/gotoblas"
./configure --libdir="-L/usr/local/openmpi,-L/usr/local/gotoblas"
./configure --libdir="-L/usr/local/openmpi" --libdir="-L/usr/local/gotoblas"
```

This option sets the Makefile variable `LIBDIR` to `<string>`. The default is to not prescribe a value as the appropriate settings are chosen – depending on the BLAS, LAPACK, METIS and UMFPACK implementations – in `Makefile.cpu.inc` and several of the Makefile templates.

`--libs=<string>`

Specifies the library loader options that are passed to the linker. The option overrides any defaults automatically determined based on the chosen build ID.

Items in `<string>` may either be separated by whitespace or commas. The option can be used multiple times on the same command line, the respective `<string>` arguments are concatenated. The following examples are equivalent:

```
./configure --libs="-lmetis -lamd -lumfpack4 -lamd -llapack -lgoto"
./configure --libs="-lmetis -lamd" --libs=-lumfpack4,-lamd,-llapack,-lgoto"
```

This option sets the Makefile variable `LIBS` to `<string>`. The default is to not prescribe a value as the appropriate settings are chosen – depending on the compiler, MPI environment and BLAS implementation – in `Makefile.cpu.inc` and several of the Makefile templates.

`--list-ids`

Prints a list of valid build IDs for the current host. No Makefile is created, no existing Makefile is touched.

Being a cross product of the tokens identifying architecture, cpu, operating system, compiler, and BLAS (and LAPACK) implementation and possibly MPI environment[8] the list can easily become lengthy. To be manageable and clear, the list uses regular expression syntax. Invoking

```
./configure --list-ids
```

on the command line of e. g. a Linux workstation with a Core2 processor would return the following list:

```
Valid build IDs (for pc-coreduo-linux32 hosts) match
against the following regular expressions:
* pc-coreduo-linux32-.*-blas.*
* pc-coreduo-linux32-.*-goto.*
* pc-coreduo-linux32-g95-.*
* pc-coreduo-linux32-gcc-.*
* pc-coreduo-linux32-intel-.*
* pc-coreduo-linux32-.*-.*-lammpi
* pc-coreduo-linux32-.*-.*-mpich2
* pc-coreduo-linux32-.*-.*-ompi

Use any of these build IDs as argument for the --id option.
The current build ID for painofsalvation is:
                pc-coreduo-linux32-intel-goto-ompi
```

Not specifying any build ID with the option `--id` means using the default build ID as determined by `guess_id` and `Makefile.buildID.inc` (see section 7 on page 41 on how a default build ID is chosen) and printed in the last line of the output. Other valid configurations are built by substituting any `.*` from the list above with valid tokens elsewhere in the list. For instance, a supported build ID based on the above list is `pc-coreduo-linux32-gcc-blas-mpich2`. In principle, any such build ID may be used as argument to the `--id` option, and if the chosen build ID is not supported yet, it can be easily added to the matching rules in `Makefile.buildID.inc`.

`--list-all-ids`

Prints a list of all valid build IDs. No Makefile is created, no existing Makefile is touched.

Being a cross product of the tokens identifying architecture, cpu, operating system, compiler, and BLAS implementation (and possibly MPI environment), the all platforms list, if printed, would be exceptionally long. To be manageable and clear, the list uses regular expression syntax. See `--list-ids` for examples. The last line of this option's output prints the default build ID of the current host.

`--list-kernel-modules`

Prints a list of all FEAST or FEATFLOW2 kernel source files the application is built from. No Makefile is created, no existing Makefile is touched.

[8] Remember that this sixth token is only required if applications are configured to be compiled for parallel execution – please note the difference to parallel (make) builds which just means compiling in parallel.

`--list-app-modules`
: Prints a list of all application source files the application is built from. No Makefile is created, no existing Makefile is touched.

`--make=<file>`
: Specifies the name of the GNU Make program to use to create the `Makefile`.

`make` needs to be invoked in order to produce a Makefile? Yes, because `configure` needs to query `Makefile.buildID.inc` for the default build ID. Internally, that is done via the system call

```
cd Featflow2/templates && make --no-print-directory -f Makefile.buildID.inc .idonly
```

As `Makefile.buildID.inc` heavily uses GNU Make language extensions (for details see section 3.1 on page 9) `<file>` must refer to (the path of) a GNU Make binary version 3.80 or higher.

`--makefile=<file>`
: Specifies the name of the Makefile to create. The default is to name it `GNUmakefile` and to rename any existing file of that name by appending the extension `.bak`. (The advantage in FEATFLOW2 context of using the file name `GNUmake` and not `Makefile` is that both the previously existing and the `configure`-based build system can be used concurrently.)

`--mode=<parallel|serial>`
: Specifies the built mode for the application, i. e. whether to compile the application for parallel (default) or serial execution.

`--modextension=<extension>`
: Specifies the extension (`<extension>`) the Fortran 90 compiler is instructed to use when creating module information files.

A Fortran 90 compiler automatically creates a module information file along with the object file for every source file. Whether to use an extension with lower or upper case letters seems to be not agreed upon. Some file operations in the Makefile require it, however, to know whether to look for a file with extension `.mod` or `.MOD`, for instance all clean up operations (`make clean`, `make purge`), but also the workaround enabled via `--movemod=YES`. This option sets the Makefile variable `MODEXTENSION` to `<extension>`. The default is to not prescribe a value as the setting is chosen in the appropriate compiler Makefile template.

`--[no-]monitor-compile-env`
: Specifies whether or not to let FEAST/FEATFLOW2's Makefiles automatically detect whether a compiler version has changed or compiler command line options have been altered for an application and subsequently prevents that object files created with different compiler versions or with differing command line options are linked. See more about this feature in section 6.2.1 on page 34.

This option sets the Makefile variable `MONITOR_COMPILE_ENV` to `YES` or `NO`. The default is to enable the feature.

`--movemod=ARG`
: Requests that module information files are forcibly moved to the object directory.

Some Fortran 90 compilers create module information files in the current working directory, even though they store object files in `OBJDIR` and do look up module information files there as well. If a compiler does not have an option to create module information files in a particular path, this option serves as a workaround. It sets the Makefile variable `MOVEMOD` to `YES` which compensates for the annoyance: Upon successful creation module information files are moved to `OBJDIR` by the Makefile. The working directory is not cluttered up with them. But there is a second, more important reason to not allow storing them in the current working directory. The approach avoids compile problems as the module information files are incompatible between different compilers. Experience has shown that the error message most compiler raise when they find a module information file created by a different compiler is not particularly clear. Being able to pinpoint the problem is more or less lucky chance for a novice and based on past experience for an advanced user. The default is to not prescribe a value as the setting is chosen in the appropriate compiler Makefile template.

`--mpiinc=<string>`
: Specifies the string that is passed to the compiler in order to add particular directories to the compiler's include path (used to search for MPI include files). The option overrides any defaults automatically determined based on the chosen build ID.

Items in `<string>` may either be separated by whitespace or commas. The option can be used multiple times on the same command line, the respective `<string>` arguments are concatenated. The following examples are equivalent:

```
./configure --mpiinc="-I/usr/local/openmpi -I/home/feastuser/openmpi"
./configure --mpiinc="-I/usr/local/openmpi" --mpiinc=-I/home/feastuser/openmpi
```

This option sets the Makefile variable `MPIINC` to `<string>`. The default is to not prescribe a value as the setting is chosen in the appropriate MPI Makefile template.

`--mpilibdir=DIR`
: Specifies the `<string>` that is passed to the linker in order to have the linker search for MPI libraries in particular directories before searching the standard directories. The option overrides any defaults automatically determined based on the chosen build ID.

Items in `<string>` may either be separated by whitespace or commas. The option can be used multiple times on the same command line, the respective `<string>` arguments are concatenated.

This option sets the Makefile variable `MPILIBDIR` to `<string>`. The default is to not prescribe a value as the setting is chosen in the appropriate MPI Makefile template.

The user should be aware that the value of `MPILIBDIR` is ignored for `MPIWRAPPERS=YES`.

`--mpilibs=<string>`
: Specifies the MPI library loader options that are passed to the linker. The option overrides any defaults automatically determined based on the chosen build ID.

Items in `<string>` may either be separated by whitespace or commas. The option can be used multiple times on the same command line, the respective `<string>` arguments are concatenated. The following examples are equivalent:

```
./configure --mpilibs="-llamf77mpi -lmpi -llam -lpthread"
./configure --mpilibs="-llamf77mpi -lmpi" --mpilibs="-llam,-lpthread"
```

This option sets the Makefile variable `MPILIBS` to `<string>`. The default is to not prescribe a value as the setting is chosen in the appropriate MPI Makefile template.

The user should be aware that the value of `MPILIBS` is ignored for `MPIWRAPPERS=YES`.

`--[no-]mpiwrappers`
: Specifies whether or not to use MPI wrapper commands for compilation and linking.

Most MPI environments provide MPI wrapper commands that ease compiling and linking by automatically adding all necessary environment variables, search directories as well as compiler and linker options. The option corresponds to a `YES` or `NO` setting of the Makefile variable `MPIWRAPPERS`. `MPIWRAPPERS=YES` triggers that the Makefile variables `MPIINC`, `MPILIBDIR` and `MPILIBS` remain unset, otherwise they need to point to a working MPI installation in order to be able to compile a FEAST application for parallel execution. The latter three variables are either set in the MPI template file (one of the files matching `Featflow2/templates/*.mk` or via the configure command line options `--mpiinc`, `--mpilibdir`, and `--mpilibs`. The default is to not use MPI wrapper commands.

`--objdir-prefix=<path>`
: Specifies that `<path>` be prepended to the path where an application's object files are stored.

  By default, an application's object files go to `object/<build ID>-<opt setting>`, i. e. `<path>=""`. On systems that enforce a disk quota, it might be necessary to store object files to some scratch area without quota restrictions. The path to the scratch area can be specified with `<path>`. Example:

  ```
  ./configure --objdir-prefix=$HOME/nobackup/feat2objectfiles
  ```

  will store all object files in a directory hierarchy under `$HOME/nobackup/feat2objectfiles` while the application binary is still created in the current working directory.

`--objdir-lib-prefix=DIR`
: Specifies that `<path>` be prepended to the path where library object files and archives are stored.

  By default, libraries are stored in `../../object/<build ID>-<opt setting>`, i. e. `<path>="../.."`. The advantage is that this setting ensures that the libraries are only compiled once and can be shared among all FEAST/FEATFLOW2 applications.[9]

  On systems that enforce a disk quota, it might be necessary to store object files to some scratch area without quota restrictions. The path to the scratch area can be specified with `<path>`. Example:

  ```
  ./configure --objdir-lib-prefix=$HOME/nobackup/feat2objectfiles
  ```

  will store all object files in a directory hierarchy under `$HOME/nobackup/feat2objectfiles`.

  Specifying a relative path here – which is the default – imposes a constraint on the directory structure for third party libraries: FEAST and FEATFLOW2 uses its own Makefile named `Makefile.FEAST` in every library directory (to only compile those parts of it that are really used and to store object files and libraries in separate object directories per build ID). The restriction is that the intended object directory must be uniformly addressable using the same prefix from any of third party library directories. Example: Storing BLAS and LAPACK in `Featflow2/librariesNEW/blas` and `Featflow2/librariesNEW/lapack`, respectively, it is feasible to use the default prefix `../..` as this leads to the object directory `Featflow2/object/<build ID>`. Storing them in `Featflow2/librariesNEW/blas` and `Featflow2/librariesNEW/lapack/3.2`, respectively, does not work as this expands to two different object directories.

`--[no-]opt[=<keyword>]`
: Specifies the level of optimisation by setting appropriate values for the Makefile variable `OPT`. The following options are valid:

  `--no-opt`
  : Disables all optimisations while including debugging symbols. When debugging code with help of a (parallel) debugger use `--no-opt`. Additionally, this option enables possibly runtime expensive FEAST code by means of instructions in `Makefile.cpu.inc` that add preprocessor instructions to `CFLAGSF90`. Currently, these are `-DENABLE_PARAMETER_CHECK` and `-DENABLE_ERROR_TRACEBACK`.[10] This is the same as specifying `--opt=no`.

  `--opt=no`
  : This is the same as specifying `--no-opt`.

  `--opt`
  : Enables all optimisations but `-ipo` for Intel compilers. This is the same as specifying `--opt=yes`.

  `--opt=yes`
  : This is the same as specifying `--opt`.

[9] FEAST application and kernel source files are written in Fortran 90. The Makefile variable `CFLAGSF90` is reserved for them. All remaining Makefile variables referring to compiler command line options – `CFLAGSF77LIBS`, `CFLAGSC` and `CFLAGSCXX` – do not contain application-specific options such that libraries can be shared by design.

[10] See section 9 on page 48

`--opt=expensive`
: Enables additional optimisations only for Intel compilers.

  Some optimisations are expensive in terms of compile time, but pay off at runtime. One of these are multifile interprocedural (IP) optimisations. Enabling them more than doubles compile time with Intel compilers[11]. When compiling the FEAST regression benchmark with this option and running the small `quicktests` test suite afterwards, it takes longer to compile than to run the benchmark.

Default is `--opt`.

`--programfile=<string>`
: Specifies this Fortran 90 source file containing the main program.

  `configure` takes the file specified as argument as root to determine recursively the minimal list of source files (by analysing the `use` statements) that need to be compiled and linked for a given FEAST application. The default value is `masterslave.f90`.

`--ranlib=<file>`
: Specifies the name of the ranlib command used to generate an index to an archive. The option overrides any defaults automatically determined based on the chosen build ID.

  This option sets the Makefile variable `RANLIB` to `<file>`. The default is to not prescribe a value as the setting is chosen in the appropriate Makefile template for the chosen operating system.

`--[no-]relink-always`
: Specifies whether or not to relink an application every time `make` is invoked.

  The option corresponds to a `YES` or `NO` setting of the Makefile variable `RELINK_ALWAYS`. The user should be aware that when developing for several different build IDs setting `RELINK_ALWAYS` to `NO` entails the risk that an application is not relinked when necessary: having re-configured the application for a different build ID, but not having changed a single source file since the application binary has been built invoking `make` will not relink the application for the current build ID. The operating system might issue a warning when trying to run it if the application binary is incompatible. But if the new build ID differs from the previous one merely in the last three tokens – to application binary is hence binary compatible –, it is left to the user to understand that the application needs relinking. The default is to relink always.

`--regression-benchmark`
: Requests that a Makefile is created controlling all aspects of the FEAST regression benchmark (compiling benchmark applications, creating scripts that run benchmark suites or selected benchmark tests). In addition, configure calls are forked in all directories starting with `src_`. This option is exclusively used by the regression benchmark.

`--srclist_app=<string>`
: Specifies a list of the source files that make up the application.

  Items in `<string>` may either be separated by whitespace or commas. The option can be used multiple times on the same command line, the respective `<string>` arguments are concatenated.

`--suppressflags=<string>`
: Specifies preprocessor and compiler command line options that should not be used for the application.

  This option redefines the Makefile variables `CFLAGSF77`, `CFLAGSF90`, `CFLAGSC`, `CFLAGSCXX` and `LDFLAGS` by removing from these variables all substrings, given as whitespace- or comma-separated list, in `<string>`. It is the converse operation of `--apponlyflags`. As such the Makefile variable `CFLAGSF77LIBS` is not touched.

[11] This holds at least true for Intel 9.x compiler releases. As the Intel 10.[0-1] compiler releases do generate wrong code for FEAST, it did not make sense to benchmark the compile time behaviour.

If an application does not compile or run properly because of specific compiler command line options and these options can, for some reason, not be globally changed in `Makefile.cpu.inc` or the Makefile templates, their usage can be inhibited by adding the exact problematic option to `<string>`. Example:

```
./configure --suppressflags="-DENABLE_USE_ONLY"
```

By default, `<string>` is empty, no compiler or linker command line option is suppressed.

`--version`
Prints `configure` version information. No Makefile is created, no existing Makefile is touched.

5.1 Implementational details

`configure` is a Perl script that is continuously expanded to meet the needs outlined in the above paragraphs. This section is intended to give an idea of what the script actually does. For details of `configure`, the reader is referred to the detailed documentation in the script itself. The first 200 lines of `configure` consist of comment lines and variable declarations. In this section of the script, lists of SbBlas, Feast kernel and architecture dependent source files are defined and assigned to constants with prefix `SRCLIST_`. Adding a source file to the kernel, renaming it or removing it from the kernel is accomplished by simply editing these lists.

Feast does not merely consist of Fortran 90 source files and adding those other files is likewise easy. Typically, and the lists are currently limited to them, these files are header files which are handled inconsistently in Feast, for several varying reasons. In Feast there are four classes of header files and every class needs a different treatment.

1. Static files containing Fortran 90 code that is needed repeatedly and therefore has been outsourced. The files are referenced by other source files using Fortran 90-style `include` statements:

   - Files containing routines with empty body that are needed to comply with interface requirements. Featflow2 does not need them because of the callback mechanisms, but Feast does it differently. Feast applications that do not call named routines include these minimal header files which avoids inflating source codes:

     ```
     kernel/user_additionalVisData.h
     kernel/user_assMatrix_band.h
     kernel/user_assMatrix_sparse.h
     kernel/user_assRhs.h
     kernel/user_exportVisData.h
     kernel/user_prepareRefinement.h
     ```

   - Files containing declarations and definitions of global variables that are used in kernel files, but have application-dependent values. Applications that do not call the corresponding kernel code still need to define the variables to maintain compilability and include the default settings provided by these files. This is a feature that is currently not used in Featflow2, but in Feast:

     `kernel/user_additionalAssVectors.h`[12]

   - Files containing an interface declaration that is used multiple times in source files. Example from Featflow2:

[12]Included by all Feast applications except `stokes`, `navsto` and `natd`. These three applications define own settings in their corresponding `userdef` modules.

$(FEASTBASEDIR)/kernel/Adaptivity/intf_hadaptcallback.inc $(FEASTBASEDIR)/kernel/DOFMa
$(FEASTBASEDIR)/kernel/DOFMaintenance/intf_coefficientMatrixSc.inc $(FEASTBASEDIR)/kern
$(FEASTBASEDIR)/kernel/DOFMaintenance/intf_coefficientVectorSc.inc $(FEASTBASEDIR)/kern
$(FEASTBASEDIR)/kernel/DOFMaintenance/intf_fbcassembly.inc $(FEASTBASEDIR)/kernel/Non
$(FEASTBASEDIR)/kernel/PDEOperators/intf_gfsccallback.inc $(FEASTBASEDIR)/kernel/PDEO
$(FEASTBASEDIR)/kernel/Postprocessing/intf_functionScBoundary2D.inc $(FEAST-BASEDIR)/kernel/Postprocessing/intf_refFunctionSc.inc $(FEASTBASEDIR)/kernel/Postprocessing
$(FEASTBASEDIR)/kernel/Postprocessing/intf_refFunctionScVec.inc $(FEASTBASEDIR)/kernel/P
$(FEASTBASEDIR)/kernel/System/cmem.inc $(FEASTBASEDIR)/kernel/Triangulation/intf_monit
$(FEASTBASEDIR)/kernel/Triangulation/intf_mshreghittest.inc

Example from Feast:

```
kernel/pboundaryValue.h
kernel/pcoeff.h
kernel/pcompSPRonBdryQ2.h
kernel/pele.h
kernel/pexactGrad.h
kernel/pexactSol.h
kernel/pfmon.h
kernel/pgetValue.h
kernel/plocGlobMapping.h
kernel/prhs.h
kernel/psearchroutine.h
```

These files have to be treated in pretty much the same way as source files defining Fortran 90 modules. They contain a C-style `#include` statement to be able to use Feast constants from `kernel/feastconstants.h` and another one for `kernel/feastcppmacros.h` as these dummy routines may be affected by preprocessor macros, too. Simply copying these files to the application's object directory does not suffice because of the C preprocessor directives involved.

`configure` automatically determines which files belong to this class by checking the blacklists associated with the header file classes 2 up to 4. If a file is not mentioned there, it is preprocessed with `cpp`, postprocessed with a small `awk` script (`bin/postprocess_cppoutput.awk`) to compensate for line number changes (see section 8 on page 45 for an explanation of what this script exactly does) and finally stored in the application's object directory.

2. Files containing C preprocessor macro definitions. They are included via C-style `#include` statements. Featflow2 source files do not currently contain them, but every Feast source files does refer to:

   ```
   kernel/feastconstants.h
   kernel/feastcppmacros.h
   ```

   These files should be listed as part of the definition of the variable `HEADER_FILES_TO_COPY` in the header section of `configure`.

3. Files containing Fortran 90 code that is dynamically created at compile time. They are referenced by other source files using Fortran 90-style `include` statements. Featflow2 does not currently have them, but Feast does refer to:

   ```
   $OBJDIR/buildconf.h
   ```

   These files do not exist at configure time, just at compile time because one of the first targets executed by a Feast Makefile, right after creating object directories, is to dynamically create these files, based on the current build ID setting. This is why `configure` has to ignore these files when determining the dependency list of source files (otherwise it would raise errors because they do not exist) and does not include them in the dependency lists in the `Makefile`. The header section of `configure` has a variable called `DO_NOT_INCLUDE` that holds the list of dynamically created files.

4. System header files. The files are referenced with Fortran 90-style `include` statements:

   ```
   mpif.h
   ```

   System header files have no dependency whatsoever on FEAST preprocessor settings, constants or data structures. As such there is no need to transfer them to the object directory.[13] These files must also be given as values of the variable `DO_NOT_INCLUDE` in the header section of `configure`.

The header section of `configure` contains two more lists of files that have not been addressed yet.

The first is assigned to the variable `MUST_INCLUDE` and takes all files that are to be included as prerequisite of every object file and that is currently empty.

The second remaining list concerns files that are automatically generated in the course of a `make` run that either builds an object file or the application itself. Currently, the list assigned to the variable `SRCLIST_GENERATED` comprises of `buildconf.h` (created by `cc_buildconf.pl`) and the nine files used by the mechanism to detect changes in the compile environment. The variable is used for setting up the clean-up targets of a FEAST `Makefile` (`clean`, `clean-libs`, `clean-app`, `purge`, `purge-libs`, `purge-app`).

Default values for the command line options are set in the subsequently defined associative array (also denominated as hash in Perl) `%defaults`. Changes are automatically communicated to `configure`'s help screen.

After the configuration section, `configure` starts with parsing command line arguments. Arguments passed to a Perl script are always stored in the variable `@ARGV`. To get them sorted and make them easily accessible, `configure` uses the Perl library `GetOpt::Long` storing them in a big hash called `%cl`. Arguments in Perl are passed by value. If the user wants to pass an argument by reference, he has to pass a reference to the variable by prefixing it with a backslash.[14] For scalars stored in a hash, e. g. the application name accessible via `$cl{'APPNAME'}`, the syntax is pretty clear:

```
\$cl{'APPNAME'}
```

Passing references to arrays to be stored in `%cl` requires a syntax that looks rather awkward: An array in a hash is still identified by its name (e. g. `CLFAGSF90`). Being an item of the hash it is addressed like a scalar variable, i. e. with `$cl{'CFLAGSF90'}`, but the programmer needs to explicitly classify the item as an array for Perl with the `@{ ... }` operator. Finally, remembering that the function `GetOptions` needs references, the expression needs a backslash prefix, as in the scalar case. Summarising, a reference to the array storing `CFLAGSF90` values is passed like this

```
\@{ $cl{'CFLAGSF90'} }
```

The difference between command line options stored as scalar values and arrays are the number of values they can take. Options like `--ar`, `--cc`, `--id`, `--make`, `--objdir-prefix`, `--opt` etc. only make sense if they take exactly one argument. If the option is given multiple times, the last occurrence counts.[15] For other options, it is convenient to be able to pass arguments additively – which automatically implies that the corresponding command line option may be specified more than once. All options used to override compiler command line flags are set up like that.

Subsequently, the command line options are checked and normalised: Errors are raised if command line options collide or use invalid keywords, options without arguments are mapped to `YES`/`NO` settings and all keywords are capitalised. Options that take multiple arguments and whose arguments can both be separated

[13] This approach should not be changed in the future as copying them become really difficult. It would require to dynamically determine where the MPI header files are located. The information might be available from `MPIINC` if the Makefile variable has been set up properly, but when using MPI wrapper commands (Makefile variable `MPIWRAPPERS=YES`) `MPIINC` is not set and portably and reliably determining the header location would become next to impossible.

[14] For the user familiar with C and C++ programming: the analogue is the `&` operator.

[15] It *does* happen frequently that options and in particular `--opt` and `--id` are passed multiple times as command line argument. Typically, a `configure` wrapper file in an application directory contains the `--opt` option. Whenever a user enforces a particular optimisation setting and invokes `./configure --opt=no`, there are two `--opt` options, one in the wrapper file, one on the command line.

by whitespace and commas are normalised by storing each value as an item of the same array. As a legacy, the environment is checked for the variable `MAKEFLAGS`. This variable is automatically set by every `make` process and contains all `make` command line options. When `configure` is called from a `Makefile`, it is obviously not possible to specify additional `configure` command line arguments on the `make` command line. But with help of the environment variable `MAKEFLAGS` it can be determined in `configure` whether particular variables have been specified on the `make` command line. Example:

```
make OPT=NO MODE=SERIAL
```

sets `MAKEFLAGS` to `-- OPT=NO MODE=SERIAL`. While examining command line options, `configure` queries this variable and if either one of the Makefile variables `FORCE_ID`, `MODE`, `MONITOR_COMPILE_ENV`, `MPIWRAPPERS`, `OPT`or `RELINK_ALWAYS` is found[16], overrides any default or command line setting. So, as a common agreement, environment variable values, passed via `MAKEFLAGS`, take precedence over command line options. If

```
make OPT=NO
```

contains a `configure` call that uses the command line option `--opt`, the value `OPT=NO` will be stored in the generated `Makefile`.

After checking command line options `configure` verifies that minimal software requirements are met, i. e. checks whether GNU Make version 3.80 or higher is available. Closely related is the test whether `make` sets the variable `MAKE`. Whenever a `Makefile` spawns sub-make processes (parallel compilation), the master and child processes need to communicate. But `make` only recognises a sub-make process as such if the sub-make process is called via `$(MAKE)`, not if called with `make`.[17] If `make` sets `$(MAKE)`, then this variable is used in the generated `Makefile` instead of the hard-coded string `make`.

In a next step, the build ID is determined. If it has not been given on the command line via the `--id` option, `Makefile.buildID.inc` is queried for it:

```
make -C../.. --no-print-directory -f Makefile.buildID.inc .idonly
```

The build ID is basically not required for `configure` to work. If it is set via command line options, the value is hard-coded in the top section of the generated `Makefile`. If it is not set on the `configure` command line, the `Makefile` will contain no build ID setting. Instead, the default build ID will be chosen in `Featflow2/Makefile.buildID.inc` whenever `make` is invoked. It is purely for user convenience that `configure` reports the build ID in its screen output. Invalid build IDs, however, cause an error.

Then, execution branches. In "benchmark mode", i. e. if the option `--regressionbenchmark` has been given, `configure` creates the `Makefile` that provides all necessary commands to compile benchmark applications and that creates benchmark control scripts (`runtests`). If a particular build ID is requested, it gets hard-coded in the `Makefile`. Subsequently, all directories with prefix `src_` are entered and a sub-configure is spawned. All command line options passed to `configure` are re-used for the sub-configure, with two changes: `--regressionbenchmark` is omitted and the (undocumented) command line options `--cached-make-program`, `--cached-gnumake-version`, `--cached-make-sets-make-var` and `--cached-build-id` are added. By means of these options, the most time-consuming part of running `configure` – the initial tests – can be skipped. The sub-configures do not question the cached values, they simply use them. The minimal software requirement tests outlined above are not redundantly run, `configure` just has to parse Fortran 90 source files, build dependency lists and create a `Makefile` according to the given settings (details are presented in the next paragraph). Being able to skip the tests reduces the time it takes to configure the benchmark by 75 %.

The remaining branch is the standard "single application mode", a `Makefile` generator for a single FEAST application. The interesting part is the source file parser creating dependency lists. It loops over all source files (specified in the constants with prefix `SRCLIST_` mentioned at the beginning of this subsection as well as the files given via the command line option `--srclist-app`)) and creates a list of all modules and include files a given file depends on. The parser supports Fortran 90-style `use` statements (including `use,only`

[16] It is a hardcoded list, `configure` does not detect every FEAST `Makefile` variable in `MAKEFLAGS`.
[17] See section 5.7.1, "How the `MAKE` Variable Works", of the GNU Make manual for further explanation.

statements), Fortran 90-style `include` and C preprocessor-style `#include` statements. Duplicate entries are ignored. The parser *does not respect conditional preprocessor directives*! Conditional preprocessor directives are evaluated at compile time, not at configure time. If a `use`, `include` or `#include` statement is nested in a `#ifdef...#endif` block, it is unconditionally put on the dependency list. If the referenced file is updated, the including source file is recompiled – even if it is not included because the preprocessor directive evaluates to a false value.[18] Header files neither mentioned in `DO_NOT_INCLUDE` nor `HEADER_FILES_TO_COPY` are gathered and stored later as entries of the Makefile variable `INCLUDED_FILES_TO_EVALUATE_WITH_CPP` to the `Makefile`. Some care is taken to correctly add or substitutes path information for the Makefile variables `SRCDIR_KERNEL`, `SRCDIR_SBBLAS`, `SRCDIR_APP` and `OBJDIR`. The related regular expression substitutions are probably not easy to understand at first glance.

The rest of the script more or less contains only hard-coded Makefile instructions.

[18] If `configure` would have to respect preprocessor directives, it would need to know about them at configure time already; changing a preprocessor directive later would require rerunning `configure` and there is no easy mechanism to enforce that.

6 Features of a Feast/Featflow2 `Makefile`

6.1 Essential Feast/Featflow2 `Makefile` Targets

Makefiles for FEAST applications are standardised due to the fact that they are automatically generated by `configure`. All provide the same set of targets. In the following, they will be outlined.

Getting help. To get a short summary of the targets most frequently used along with a list of Makefile variables most frequently explicitly overridden on the command line, it takes a

```
make help
```

Compiling and file removal. The most popular FEAST Makefile target is without doubt the one that triggers the compilation of all source files and finally links the application:

```
make <feastappname>
```

To reduce typing efforts, it has three aliases:

```
make default
make all
```

or simply

```
make
```

Closely related is a target that enforces compilation of all source files without compiler optimisations:

```
make debug
```

is essentially the same as

```
make OPT=NO
```

and is typically used in every day's code development.

There is another "trick" that helps saving (compile) time when doing (low-level) kernel development: Compiling a FEAST application from scratch takes a while and editing FEAST files at the lower end of the Fortran 90 module hierarchy triggers recompilation of almost every source file as these basic modules are (directly or indirectly) used by every high-level FEAST source file. Chasing bugs in those low-level files is hence rather time consuming (edit, make, run, study the output, edit, make, run, study ...). At least the `make` process can be sped up – if and only if no compiler optimisations are performed. In the latter case there is not much one can do about it because recompilation of source files higher in hierarchy must not be skipped because of interprocedural and -file optimisations ("inlining"). Everyone not being a compiler developer and has hence insufficient insight into how the compiler performs optimisations should refrain from bypassing recompilation of some source files and get instead a cup of coffee while waiting for the optimised application to be built. For unoptimised builds the story is different. If the changes do not comprise any interface change, it is feasible to merely recompile the source file that was changed and then to relink the application. If source files higher in hierarchy are *not* recompiled compile time is drastically reduced. It could be done manually by invoking the necessary compile and link commands, but also with help of a some FEAST Makefile targets which is definitely more convenient. For every source file there is a rule to create a corresponding object file (for the chosen build ID) and there is a separate rule to link the application. It is essential not to skip the linking, as the make targets for a simple object file do not delete and/or replace the executable. When editing e. g. the module `fsystem` it suffices to issue a

```
make fsystem.o link
```

to be able to test the changes in one's application. The command will compile only the source file `fsystem.f90` and then link the FEAST application. When requesting compilation of an object file a little higher in hierarchy, e. g. `io.o`, the behaviour is a bit different. Because `fsystem.o` has only the dependency `fsystem.f90`, other source files have more:

```
make io.o link
```

will not only compile `io.o` if `io.f90` is newer, but will also compile any subordinate source file that has been updated. Because `make` checks whether dependencies of `io.o` have a newer time stamp and recompiles them if that is the case. Then, `io.o` is recompiled from `io.f90` and the module information files of the subordinate modules. Upon completion, the second requested target, `link`, is executed which, as in the example above, links the application, using newly compiled object files up to `io.o` and reusing old versions for the remaining object files.

To delete the application binary, all object files and all libraries for the current build ID, invoke

```
make clean
```

Often, it is not necessary to recompile the third-party libraries shipped with FEAST. Remembering that it also takes a while to compile them, it seems natural to provide a target that only removes the application binary and FEAST object files, but to keep all libraries.

```
make clean-app
```

caters for this need.

Sometimes it is necessary to separately compile the third-party libraries shipped with FEAST. This is e. g. the case when investigating their optimal compiler settings or testing a new library version. The build ID determines which libraries need to be compiled; the list is stored in the Makefile variable `BUILDLIB` whose value can conveniently be queried via `make id` (see below). For every library there exists a (dynamic) target of that name to build that particular library. Let us assume without loss of generality the following setting:

```
BUILDLIB = metis amd umfpack4 lapack
```

Then, `make` will accept the Makefile targets `amd`, `lapack`, `metis`, and `umfpack4`. For convenience, the following aliases exist:

```
make metis     =  make libmetis     =  make libmetis.a
make umfpack   =  make libumfpack   =  make libumfpack.a
...
```

To build all libraries, the user can use

```
make libs
```

and to remove all libraries along with the object files they are built from issue

```
make clean-libs
```

For more fine-grained control specialised cleanup targets exist: `clean-amd`, `clean-blas`, `clean-gpu`, `clean-lapack`, `clean-metis`, `clean-umfpack` etc.

Makefile targets with prefix `clean` are associated with the current build ID. Analogous targets with prefix `purge` act as if the `clean` variant were given for all possible build IDs. Example: Developing for the build IDs `pc-opteron-linux64-intel-goto[-ompi]`, `pc-opteron-linux64-sunstudio-goto[-ompi]` and `pc-opteron-linux64-pgi-goto[-ompi]` (parallel and/or serial execution) accordingly named object directories will be created:

```
object/pc-opteron-linux64-intel-goto-ompi-optNO
object/pc-opteron-linux64-intel-goto-ompi-optYES
object/pc-opteron-linux64-intel-goto-optNO
object/pc-opteron-linux64-sunstudio-goto-ompi-optYES
object/pc-opteron-linux64-pgi-goto-optYES
...
```

To remove these directories, one could invoke

```
make clean ID=pc-opteron-linux64-intel-goto-ompi MODE=PARALLEL OPT=NO
make clean ID=pc-opteron-linux64-intel-goto-ompi MODE=PARALLEL OPT=YES
make clean ID=pc-opteron-linux64-intel-goto MODE=SERIAL OPT=NO
make clean ID=pc-opteron-linux64-sunstudio-goto-ompi MODE=PARALLEL OPT=YES
make clean ID=pc-opteron-linux64-pgi-goto MODE=SERIAL OPT=YES
...
```

but the following command suffices:

```
make purge
```

The targets `purge-app`, `purge-libs`, `purge-amd`, `purge-blas`, `purge-gpu`, `purge-lapack`, `purge-metis`, `purge-umfpack` work alike.

Most users prefer to store object files in some scratch area instead of in a subdirectory of their working copy. In the latter case, it is rather straightforward to remove a single object file. Using the shell's tab expansion features it only takes a few keystrokes to expand the fully qualified path name of the object file:

```
rm -f object/pc-opteron-linux64-intel-goto-ompi-optNO/io.o
```

But even then the user needs to remember the current build ID. When object files are stored at some other location (and it is likely that this location varies on different architectures (see configure options `--objdir-prefix` and `objdir-lib-prefix`, page 23), the risk to accidentally delete a wrong file increases, as removal becomes more cumbersome. The Makefile knows where an object file is created, so why not let it deal with its deletion as well? That is the intention of the target `delete`. It takes as an argument the filename (without path information) and deletes it in the object directory set up for the current build ID. Example:

```
make delete FILE=io.o
```

Getting information. A FEAST Makefile contains four targets that merely provide information to the user.

```
make id
```

outputs a comprehensive overview of all Makefile settings that affect compilation. The report includes the build ID and the paths to compilers and preprocessors (FEAST and C). If any of them can not be found, a warning is shown instead. Subsequently, the list comprises all relevant Makefile variables and their setting. This report is very helpful for debugging compile problems.

```
make idonly
```

merely prints the current build ID. The target is e. g. used by the nightly regression benchmark script to include build ID information in the report e-mail.

```
make list-ids
```

prints a list of valid build IDs for the current host. It is equivalent to the invocation of `./configure --list-ids`.

```
make list-all-ids
```

prints a complete list of valid build IDs. It is equivalent to the invocation of `./configure --list-all-ids`.

Overriding Makefile variables. A variable defined on the command-line will take precedence over the same variable defined in a Makefile. This general Makefile convention is usually exploited in FEAST for purposes as follows:

Often, a user needs to study the effect of changes to his code and runs a script that performs a whole battery of tests in a loop over several application binaries. Instead of compiling and renaming these applications or editing the local configure wrapper script where this setting is hardcoded, the user simply overrides the application name on the command line:

```
make APPNAME=poisson-test1
```

edits his code and compiles the next variant with

```
make APPNAME=poisson-test2
```

and finally starts his script that subsequently invokes the binaries `poisson-test1` and `poisson-test2` to perform the tests.

Other Makefile variables which are naturally overridden on the command line include `OPT`, `MODE`, `MPIWRAPPERS`, `RELINK_ALWAYS`, and `MONITOR_COMPILE_ENV`. See section 5 on page 12ff to learn about valid values and the resulting behaviour.

6.2 Automatic sanity checks

Every time a Makefile target that involves compilation is executed, a number of sanity checks are performed first. The intention is to automatically catch as many latent problems as possible before they can cause trouble. These checks comprise e. g.

- (only in FEAST:) invocation of `feast/feast/Makefile.symlinks` by calling the target `symlinks` present in every FEAST Makefile:

  Files that – in an attempt to reduce redundancies – are realised via symbolic links are not stored in the CVS repository. Given that they may be dependencies of rules that need to be executed for the requested target, a submake process is invoked that processes the instructions in `Makefile.symlinks`: Any object from a list of file system objects that does not exist in the current working copy of FEAST is created as symbolic link to a given file system object, hard coded in the header section of `Makefile.symlinks`.

  In case the target of a symbolic links does not exist, an error message is raised. If the target is a file, it reads as follows:

  ```
  # Makefile.symlinks: File <[name of target]> does not exist!
  # Makefile.symlinks: Restore of symbolic link <[name of symbolic link]> failed.

  # Makefile.symlinks: Ensure that none of the following holds:
  # Makefile.symlinks: * the file has been lost;
  # Makefile.symlinks: * the file has never been committed to the CVS repository;
  # Makefile.symlinks: * your working copy is not up to date;
  # Makefile.symlinks: * you checked out individual older revisions from the CVS
  # Makefile.symlinks:   repository (willingly risking an inconsistent working copy).
  # Makefile.symlinks:   In this case, it might be that the file has had not been
  # Makefile.symlinks:   added yet to the CVS repository at the date of the older
  # Makefile.symlinks:   revisions you selected.
  ```

  If the missing target of the symbolic link to be created is a directory, the error message is similar.

- reassurance that the current build ID is handled fully and correctly by `Makefile.cpu.inc`.

  As defined in section 4 on page 11 a build ID consists of five (serial application) or six tokens (parallel application). Each token triggers the setting of related Makefile variables in `Makefile.cpu.inc` or one of the Makefile templates. Failure to set them usually has catastrophic effects for the compilation process (compiler not found, invalid compiler command line options etc.). That is why every time a token has been handled in `Makefile.cpu.inc` or the Makefile templates a corresponding Makefile variable is set. If the first token has been matched, `TOKEN1` is set, for the second token `TOKEN2` etc. up to `TOKEN6`. If any of these Makefile variables is found to be unset after `Makefile.cpu.inc` has been parsed[1], an error is raised explaining which tokens are unset. The user is requested to update the matching rules in `Makefile.cpu.inc`. This approach catches both incomplete and missing rules and build IDs.

- confirmation that the chosen build ID matches the current architecture:

  A user can accidentally enforce a particular build ID (e. g. `pc-opteron-linux64-intel-goto-mpich`), forget about this fact in the course of time and later try to compile on a different architecture (for instance as Solaris Sparc system, `sun4u-sparcv9-sunos-*-*-*`). In such a case an error message is raised because the environment and compiler names and compiler settings are typically completely different and incompatible for different build IDs. In this example the error message would read:

  ```
  # Build ID pc-opteron-linux64-intel-goto-mpich is *not* valid
  # for current host!
  # ../../bin/guess_id reported a host of type sun4u-sparcv9-sunos.
  # Cowardly refusing compilation.
  make: *** [verify-id] Error 1
  ```

  When cross-compiling it is necessary to disable this setting. It is done by either adding the option `--force-id` to `configure` or by overriding the Makefile variable `FORCE_ID` on the command line and setting it to `YES`.

- verification that certain Makefile variables have exactly one out of a set of possible values:

  The Makefile variable `OPT` takes either the values `NO`, `YES` or `EXPENSIVE`. If that requirement is not met, an error message is raised and all compiler definitions are unset to render compilation impossible. The same happens if `MODE` is neither set to `PARALLEL` nor `SERIAL`.

- scrutiny for conflicting compile options:

  It does not make sense, but it is perfectly possible to set the Makefile variable `MODE` to `PARALLEL` while including the preprocessor directive `-DENABLE_SERIAL_BUILD` in some of the Makefile variables storing compiler command line options (`APPONLYFLAGS`, `CFLAGSF77`, `CFLAGSF90`, `CFLAGSC`, `CFLAGSCXX`). Such attempts raise an error as well.

There is yet another sanity check implemented that verifies compilers, compiler versions and compiler command line options. The mechanism is a bit complex which is why a separate subsection is dedicated to it.

6.2.1 A mechanism to detect changes in the compile environment

Makefiles keep track of dependencies among the files of a project. But object files do not merely depend on their corresponding source files and their dependencies. They also depend on the compiler vendor, on the compiler version and on command line options to the compiler as well as on external libraries such as BLAS, LAPACK or the MPI implementation. Different compilers are dealt with in FEAST using different

[1]Which is after the according `include` statement.

`build IDs`.[2] (To learn more about `build IDs` see section 4 on page 11.) Compiler versions and command line options to the compiler are a different story. There is no general approach to determine from a given object file the compiler version it has been created with. Compiler releases are usually not 100 % compatible, in particular not when it comes to code optimisations. Applications should be compiled and linked with one particular version, not mixed. This basic rule also holds for Fortran 90 modules. `make` by itself is of no help when it comes to enforcing this rule, nor is there any other standard tool that deals with the issue. If a system administrator (silently) updates a compiler, or if the user experiments with different releases and does not pay thorough attention during incremental recompilation of a FEAST application, phantom (compile, link, or runtime) errors can be expected.

Even more difficult to retrieve from a given object file are the command line options used to compile this very object file. In particular, there is no general way to find out whether a given kernel module object file has been compiled with a particular preprocessor macro or not. It has been stated earlier that the FEAST kernel heavily uses preprocessor directives for both performance and portability issues. Code fragments are enclosed in

```
#ifdef FOO
...
#endif
```

to be included in the executable only when explicitly requested by specifying the preprocessor macro `-DFOO` as compiler (read: preprocessor) command line argument. Procedure parameters, for instance, are only thoroughly checked for validity for non-optimised builds, executables for production runs omit such tests as they are supposed to be performance-tuned. Preprocessor directives controlled by a single preprocessor macro are usually restricted to one kernel module[3], but there are some preprocessor macros that influence multiple kernel modules: `ENABLE_SERIAL_BUILD`, `ENABLE_ERROR_TRACEBACK`, `ENABLE_FORTRAN_ALLOCATE`, `ENABLE_PARAMETER_CHECK`, `ENABLE_BUILTIN_MPI_GATHER` and a few more. In most cases, it may work to set one of these preprocessor macros when compiling one module and not setting it for another. The application may compile, link and even run properly. But it is easy to see that in general one wants preprocessor macros being consistently used.

A software package typically leaves it to the user that both goals are met, i. e. the consistent usage of a single compiler version and one set of preprocessor macros throughout all source files of an application. FEAST `Makefiles` aim to provide a safer and more convenient environment: Changes are detected automatically. This mechanism is – to the best of our knowledge – unique. The associated instructions are enabled for `MONITOR_COMPILE_ENV=YES`. The Makefile target `check-or-store-settings` is an unconditional dependency for every object file and as such gets executed (once) *before* any source file is compiled. It is checked whether a previous compilation for the current build ID has stored compiler modification dates and compiler settings in the object directory. If this information is available, the current settings are compared against them. As the compile system stores library object files and archives (`*.o` and `*.a` files) in an object directory shared among all FEAST applications and stores all kernel and application's object files in a separate object directory per application, this check is done twice: first for the application's object directory (`OBJDIR`), then for the libraries' object directory (`OBJDIR_LIB`). Any difference between stored and current settings lets the `make` process abort. The user is presented with an error message similiar to the one shown in example 6.1: Previously, the Fortran 90 compiler command line options only comprised of

```
  -O0 -g -DUSE_COMPILER_INTEL
```

while the current compilation adds one more preprocessor macros:

```
  -DENABLE_PERFCOUNTER_USER_1
```

[2]It is still possible, though, to use a wrong compiler when using MPI wrapper commands. A Makefile prepared for build ID `pc-opteron-linux64-intel-goto-ompi` uses the first `mpif90` command it finds in `$PATH`. If this `mpif90` has been set up to use e. g. the Pathscale Fortran 90 compiler, a FEAST Makefile does not catch it. The compiler will most likely issue some warnings about unknown command line options.

[3]e. g. `ENABLE_DEBUG_STORAGE` and `DISABLE_STORAGE_PADDING` are exclusively used in module `storage`, `ENABLE_HLAYER_DEPENDENT_TRANSFER` does only affect module `transfer` etc.

```
# Checking whether compiler or compiler command line settings
# used to build the application have changed since last time
# 'make' was run for this build ID.
5c5
< CFLAGSF90 = -O0 -g -DUSE_COMPILER_INTEL
---
> CFLAGSF90 = -O0 -g -DUSE_COMPILER_INTEL -DENABLE_PERFCOUNTER_USER_1
#
# Compiler settings have changed.
# The above output is 'diff' output. Syntax:
# < [old settings]
# ---
# > [new settings]
#
# FEAST application should be recompiled.
# Please issue a
# % make clean-app; make obj
# or if you prefer tabula rasa
# % make clean; make
#
# You can deactivate this checking by compiling
# with 'MONITOR_COMPILE_ENV=NO'.
########################################################################
make[1]: *** [check-settings-objdir] Error 1
```

Listing 6.1: `make` has detected that compiler command line options have changed.

In order to use one common set of preprocessor macros per FEAST application (and build ID), the user is advised to recompile the application. Minimally via

```
make clean-app; make obj
```

or – for the more paranoic user – more rigorously via

```
make clean; make
```

which will recompile all libraries, too.

The case where the compiler has been updated (since the last time `make` was called for the current build ID) is caught as well. An error message as depicted in example 6.2 is raised.

At the end of the error message it is stated how to overcome this error message *without* recompilation. A user that is experienced in FEAST and is fully aware of the implications can override the Makefile variable `MONITOR_COMPILE_ENV` explicitly on the command line[4] by adding `MONITOR_COMPILE_ENV=NO` to the command he invoked before, e. g.

```
make -j 2 benchmark MONITOR_COMPILE_ENV=NO
```

A setting of `NO` lets the Makefile target `check-or-store-settings` evaluate to an empty rule. This restores default Makefile behaviour where consistency of compiler, compiler version and compiler command line options is not guaranteed.

[4]The rule for Makefile variables to remember is (like the old paper, scissors, rock game): command-line beats Makefile beats environment. A variable defined on the command-line will take precedence over the same variable defined in a Makefile which will take precedence over the same variable defined in the environment.

```
# Checking whether compiler or compiler command line settings
# used to build the application have changed since last time
# 'make' was run for this build ID.
#
# Version information of F77 compiler has changed:
# previously: G95 (GCC 4.0.3 (g95 0.91!) Oct 30 2006)
# currently : G95 (GCC 4.0.3 (g95 0.91!) Nov 29 2007)
#
# File information of F77 compiler has changed:
# previously: -rwxr-xr-x 1 root root 251640 Oct 31 2006 /usr/local/g95/2006-10-30/bin/g95
# currently : -rwxr-xr-x 1 root root 220720 Nov 30 07:43 /usr/local/g95/2007-11-29/bin/g95
#
# Version information of F90 compiler has changed:
# previously: G95 (GCC 4.0.3 (g95 0.91!) Oct 30 2006)
# currently : G95 (GCC 4.0.3 (g95 0.91!) Nov 29 2007)
#
# File information of F90 compiler has changed:
# previously: -rwxr-xr-x 1 root root 251640 Oct 31 2006 /usr/local/g95/2006-10-30/bin/g95
# currently : -rwxr-xr-x 1 root root 220720 Nov 30 07:43 /usr/local/g95/2007-11-29/bin/g95
#
# FEAST application should be recompiled.
# Please issue a
# % make  clean-app; make  obj
# or if you prefer tabula rasa
# % make  clean; make
#
# You can deactivate this checking by compiling
# with 'MONITOR_COMPILE_ENV=NO'.
########################################################################
make[1]: *** [check-settings-objdir] Error 1
```

Listing 6.2: `make` has detected that another compiler is being used.

6.2.2 Implementational details

This section deals with the following questions: Why is the mechanism to detect changes in the compile environment implemented in the way it is done? What exactly is stored/compared, where is it stored and what are the odds and ends to be respected?

One might innocently argue that it is sufficient to put the files with Makefile settings on the prerequisite list of every object file. Whenever `Makefile.cpu.inc` or one of template files matching `Featflow2/templates/*.mk` changes, an application should be recompiled. This simple approach has a few drawbacks: Firstly, an application would get recompiled unnecessarily most of the time. If someone changes an optimisation option for compiler A and commits these changes to the repository, it should not have any consequences for those not using compiler A. But a Makefile that simply looks at the timestamps of `Makefile.cpu.inc` and all template files would trigger the recompilation of any FEAST application as soon as the repository changes have been merged into the working copy. Secondly, and that is worse, this approach might overlook conditions where a recompilation is indeed necessary. It is possible to override any Makefile variable with a setting passed along on the command line. `Makefile.cpu.inc` and/or one of the template files matching `Featflow2/templates/*.mk` might define

```
  CFLAGSF77 = -DUSE_COMPILER_INTEL -i4 -assume underscore
```

but invoking

```
  make CFLAGSF77="-DUSE_COMPILER_INTEL -i8 -assume 2underscores"
```

is perfectly allowed. The command would override the `CFLAGSF77` setting and recompile merely source files that have changed, if any. In the best case, the command would result in a linker error. Why? The timestamp of `Makefile.cpu.inc` or of any of the template files matching `Featflow2/templates/*.mk` does not change by overriding a Makefile variable on the command line. The change would go undetected. Depending on the differences between the setting provided on the command line and the one stored in `Makefile.cpu.inc` or used previously, the linker might not find some symbols and quit with an error.

These two drawbacks could be caught by storing the values of some set of Makefile variables to a file and compare against them later. FEAST has a Makefile target `id` summarising preprocessor, compiler, linker, compiler and linker settings and object directories. The output of

```
  make id
```

could be stored. Any subsequently called `make` could compare its current values against these stored ones. Unfortunately, there are two drawbacks to this strategy, too. First of all, `make id` lists more information than necessary. The summary comprises linker settings (`LD`, `LIBDIR`, `LIBS`, `MPILIBDIR`, `MPILIBS`) which, when changed, do not require source files to be recompiled. The application needs to be merely relinked. Then, there is path information contained in the summary: the values of `OBJDIR` and `OBJDIR_LIB` are contained in the strings reported for `CFLAGSF90` and `LIBDIR` - as arguments to the `-moddir` and `-L` options, respectively. Storing this information in the application's object directory would not give rise to any problem. But storing them to the libraries' object directory would; recall that object files and static library files of libraries like BLAS, LAPACK, UMFPACK etc. are shared among FEAST applications. The reason is that they have no dependency on FEAST and – if not using system-provided versions but the reference implementations shipped with FEAST – take quite a while to compile. Storing path information on the current application's working directory to the (shared) libraries' object directory would give a conflict when compiling several FEAST applications. The FEAST regression benchmark consists of several applications. When trying to compile the benchmark, the following situation would arise: Having compiled application `bouss2dmini` the libraries' object directory would contain a file with content similar to

```
  [...]
  CFLAGSF90  = [...] -moddir=/home/feat2user/nobackup/feat2objectfiles/benchmark/bouss2dmini/object/pc-optero
linux32-gcc-goto-optYES [...]
[...]
  LIBDIR     = [...] -L/home/feat2user/nobackup/feat2objectfiles/benchmark/bouss2dmini/object/pc-opteron-lin
```

```
gcc-goto-optYES -L/home/feat2user/nobackup/feat2objectfiles/object/pc-opteron-linux32-gcc-goto-\
optYES [...]
[...]
  OBJDIR     = /home/feat2user/nobackup/feat2objectfiles/benchmark/bouss2dmini/object/pc-opteron-linux32-gcc-goto-\
ompi-optYES
  OBJDIR_LIB = /home/feat2user/nobackup/feat2objectfiles/object/pc-opteron-linux32-gcc-goto-optYES
```

When `make` proceeds to compile e. g. `cc2d` the values of the above listed Makefile variables as reported by `make id` would be:

```
  CFLAGSF90  = [...] -moddir=/home/feat2user/nobackup/feat2objectfiles/benchmark/cc2d/object/pc-opteron-\
linux32-gcc-goto-optYES [...]
  LIBDIR     = [...] -L/home/feat2user/nobackup/feat2objectfiles/benchmark/cc2d/object/pc-opteron-linux32-\
gcc-goto-optYES -L/home/feat2user/nobackup/feat2objectfiles/object/pc-opteron-linux32-gcc-goto-\
optYES [...]
  OBJDIR     = /home/feat2user/nobackup/feat2objectfiles/benchmark/cc2d/object/pc-opteron-linux32-gcc-goto-\
ompi-optYES
  OBJDIR_LIB = /home/feat2user/nobackup/feat2objectfiles/object/pc-opteron-linux32-gcc-goto-optYES
```

The source code of the two applications are stored in different directories and this difference is reflected in differently named object directories. Consequently, values referencing these object directory names collide.

So, the output of `make id` can not be used to determine automatically whether a source file needs to be recompiled because of changed compiler settings or an updated compiler. Removing path information from its output and reducing the list to the minimum by omitting unnecessary settings (linker etc.) is the path to follow.

The compiler command line settings that are stored are:

```
  CFLAGSF77LIBS = [...]
  CFLAGSF77     = [...]
  CFLAGSF90     = [...]
  CFLAGSC       = [...]
  CFLAGSCXX     = [...]
  APPONLYFLAGS  = [...]
  INC           = [...]
```

Any path information is removed from these settings with help of a small `sed` script.

Apart from the file containing all compiler command line settings, eight more files are created in the current object directory (`OBJDIR`, easily to be determined from the `make id` output), i. e. two files per compiler (Fortran 77, Fortran 90, C and C++). For every compiler the output of

```
  ls -lLd <absolute_path_to_compiler>'
```

is stored in one file. To a separate file, the output of

```
  <compiler> <get-version-info>
```

is stored. As the syntax to retrieve the compiler version is different for most compilers, the above command effectively boils down to an evaluation of the Makefile variables `F77VERSION`, `F90VERSION`, `CCVERSION` and `CXXVERSION`. They are defined in one of the files matching `Featflow2/templates/*.mk`.

In total, these nine files catch all cases where a compiler has been updated, swapped for another and similar conditions as well as every case where compiler command line settings are altered.

Feast Makefiles are parallel-safe. But that is a property one has to work hard for, it is not for free. Compiling multiple Feast applications simultaneously – which is what happens when compiling the Feast regression benchmark in parallel – means potentially concurrent read and/or write access to the settings files in the libraries' object directory (`OBJDIR_LIB`). To guarantee that only one process is accessing the above mentioned nine files in an object directory, a lock file mechanism is implemented. If a lock file is found, a process waits for a certain amount of time[5] for the lock to be released. In case the lock has not been released after (currently) ten iterations, compilation aborts with an according error message. By trapping the signals SIGINT (Control-C pressed), SIGQUIT (core dump) and SIGKILL (process killed), the lock file is automatically released at abnormal termination of Makefile execution.

The mechanism to detect changes in the compile environment implies and enforces a side condition: The values of `CFLAGSF77LIBS`, `CFLAGSF77`, `CFLAGSF90`, `CFLAGSC`, `CFLAGSCXX` and `INC` must not differ among FEAST applications. They are stored to the libraries' object directory (`OBJDIR_LIB`). Differing values would result in conflicts if `MONITOR_COMPILE_ENV` is set to `YES` which is in particular a problem when compiling the FEAST regression benchmark. The only Makefile variable that may vary between FEAST applications is `APPONLYFLAGS`. With this variable the user can pass along options that are somehow specific for a FEAST application. They are neither used to compile libraries nor stored to the object directory where libraries are stored. See the file `fbenchmark2/master.configure` for example usage.

[5]Currently, this interval is set to one second.

7 `Makefile.inc`, `Makefile.buildID.inc`, `Makefile.cpu.inc` and Makefile templates

The files `Makefile.inc`, `Makefile.buildID.inc` and `Makefile.cpu.inc` have been mentioned a few times already, as well as FEAST's Makefile templates. It is about time to shed some light on this part of FEAST's compile system. A FEAST `Makefile` contains dependency lists for every source file, rules to compile source files, sanity check instructions - and an `include` statement for `Makefile.inc` which in turn includes both `Makefile.buildID.inc` and `Makefile.cpu.inc` and the latter refers to the Makefile templates. These files are automatically parsed prior to compilation as explained in section 6.2.

`Makefile.cpu.inc` and the Makefile template files matching `Featflow2/templates/*.mk` provide compiler names, compiler command line options, instructions to extend include and library search paths[1], linker options etc. The flexibility of FEAST's Makefiles with respect to supporting differing platforms, compilers, compiler releases, BLAS (and LAPACK) implementations, and MPI environments is mainly due to these files and their ability to translate a build ID into appropriate compile settings. FEAST's custom preprocessor which also realises portability to a big extend is discussed in the next section 8.

`Makefile.inc` merely contains two include statements for `Makefile.buildID.inc` and `Makefile.cpu.inc` and is provided for backwards compatibility only. `Makefile.buildID.inc` and `Makefile.cpu.inc` are a big collection of conditionals and variable definitions. Right at the beginning of the former file, a script called `guess_id` is invoked to determine the machine/architecture, the CPU type and the operating system. These tokens form the first half of a FEAST build ID. The script is inspired by GNU `config.guess`, but in contrast to that script it distinguishes more among the various Intel and AMD processors (by looking at cpu family and model numbers). Being able to distinguish these is crucial to be able to tune compiler command line options to fully exploit a processor's potential.

In the next section of `Makefile.buildID.inc`, the three tokens are expanded to a fully qualified build ID. A default build IDs for every architecture FEAST has been ported to is defined this way. Note that any particular build ID, either given via the configure command line options `--id` (see page 18) or on the make command line via `ID=<build ID>`, will override these defaults.

In `Makefile.cpu.inc` initial values for compiler commands and compiler command line options are set. Compiler and linker commands are overridden later when a conditional matches the build ID, but to catch the opposite case of an unhandled build ID `F77`, `F90`, `CC`, `CXX`, and `LD` are set up to raise an error message that the compiler has not been defined. When trying to compile, the user will be informed that

```
No Fortran 77 compiler specified.
```

Compiler command line options are set to an empty value, with the exception of Fortran 90 compiler command line options, stored in `CFLAGSF90`. For unoptimised builds preprocessor directives are unconditionally enforced that activate additional code to perform runtime-expensive parameter checks and enable FEAST's error traceback mechanism (`-DENABLE_PARAMETER_CHECK` and `-DENABLE_ERROR_TRACEBACK`). For both optimised and unoptimised builds, `CFLAGSF90` is extended by all preprocessor directives found in the file `kernel/feastcppswitches.h` (FEAST) or `feat2cppswitches.h` (FEATFLOW2). The file is parsed, any non-empty line that does not start with a hash character ('#') gets a '-D' prepended and is then added to `CFLAGSF90`. The exemplary content of this file,

[1]Paths used to search for module files and include files.

```
ENABLE_DEBUG_VISOUTPUT

#ENABLE_ENHANCED_KERNEL_STATISTICS

ENABLE_USE_ONLY
```

is translated into the following setting

```
CFLAGSF90=-DENABLE_DEBUG_VISOUTPUT -DENABLE_USE_ONLY
```

for optimised builds and

```
CFLAGSF90=-DENABLE_PARAMETER_CHECK -DENABLE_ERROR_TRACEBACK -DENABLE_DEBUG_VISOUTPUT
          -DENABLE_USE_ONLY
```

for unoptimised builds.[2]

`Makefile.cpu.inc` consists of conditionals that map a build ID or parts of it to Makefile settings. Whenever settings are used multiple times, i. e. for several build IDs, they are stored in a separate file with extension `mk` in the subdirectory `templates` and a mere reference to it is stored in `Makefile.cpu.inc`. This procedure ensures that settings are consistent, reduces redundancy and eases maintainability. Example 7.1 illustrates how 24 possible build IDs for a Core2 chip are efficiently defined.

[2]No precautions whatsoever are taken when parsing `kernel/feastcppswitches.h` or `feat2cppswitches.h`, respectively, and adding strings to CFLAGSF90. One might argue that this is a security hole as it is perfectly possible and fairly easy to set up the file in such a way that arbitrary code is executed. (e. g. a line like
`FOO; rm -rf $HOME; echo`
would destroy the user's file space as soon as he tries to compile Feast.) But the user already trusts `configure` and the generated `Makefiles` which also could do harmful operations. There is no point spending much effort to strip semicolons etc. from the file or to only accept certain keywords and discard all others if there are so many other possible ways to damage a user's file space.

```
ifeq ($(call match,$(ID),pc-coreduo-linux32-g95-.*),yes)
include $(TEMPLATESDIR)/pc-generic.mk
include $(TEMPLATESDIR)/g95-generic.mk
ifeq ($(call optimise), YES)
CFLAGSF77LIBS := -march=nocona $(CFLAGSF77LIBS)
CFLAGSF77     := -march=nocona $(CFLAGSF77)
CFLAGSF90     := -march=nocona $(CFLAGSF90)
CFLAGSC       := -march=nocona $(CFLAGSC)
LDFLAGS       := -march=nocona $(LDFLAGS)
endif
endif

ifeq ($(call match,$(ID),pc-coreduo-linux32-intel-.*),yes)
include $(TEMPLATESDIR)/pc-generic.mk
include $(TEMPLATESDIR)/intel-generic.mk
ifeq ($(call optimise), YES)
CFLAGSF77LIBS := -axT $(CFLAGSF77LIBS)
CFLAGSF77     := -axT $(CFLAGSF77)
CFLAGSF90     := -axT $(CFLAGSF90)
CFLAGSC       := -axT $(CFLAGSC)
LDFLAGS       := -axT $(LDFLAGS)
endif
endif

ifeq ($(call match,$(ID),pc-coreduo-linux32-pgi-.*),yes)
include $(TEMPLATESDIR)/pc-generic.mk
include $(TEMPLATESDIR)/pgi-generic.mk
ifeq ($(call optimise), YES)
CFLAGSF77LIBS := -tp core2 $(CFLAGSF77LIBS)
CFLAGSF77     := -tp core2 $(CFLAGSF77)
CFLAGSF90     := -tp core2 $(CFLAGSF90)
CFLAGSC       := -tp core2 $(CFLAGSC)
endif
endif

ifeq ($(call match,$(ID),pc-coreduo-linux32-sunstudio-.*),yes)
include $(TEMPLATESDIR)/pc-generic.mk
include $(TEMPLATESDIR)/sunstudio-generic.mk
endif

ifeq ($(call match,$(ID),pc-coreduo-linux32-.*-atlas.*),yes)
include $(TEMPLATESDIR)/atlas-generic.mk
endif

ifeq ($(call match,$(ID),pc-coreduo-linux32-.*-goto.*),yes)
include $(TEMPLATESDIR)/gotoblas-generic.mk
endif

ifeq ($(call match,$(ID),pc-coreduo-linux32-.*-.*-lammpi),yes)
include $(TEMPLATESDIR)/lammpi-generic.mk
endif

ifeq ($(call match,$(ID),pc-coreduo-linux32-.*-.*-mpich2),yes)
include $(TEMPLATESDIR)/mpich2-generic.mk
endif

ifeq ($(call match,$(ID),pc-coreduo-linux32-.*-.*-ompi),yes)
include $(TEMPLATESDIR)/openmpi-generic.mk
endif
```

Listing 7.1: Excerpt from `Makefile.cpu.inc` defining build IDs for Core2 chips

8 Feast's preprocessor `f90cpp`

The definition of a preprocessor is a program that processes its input data to produce output that is used as input to another program. The language standards for e. g. C and C++ define such a preprocessor, the Fortran 90 language standard does not. Nonetheless, many Fortran 90 compilers provide one and FEAST highly benefits from its use. The additional flexibility provided by preprocessor directives has been outlined earlier (see section 3 on page 8). But without the supportive corset of a standard, a variety of different flavours and syntaxes of Fortran 90 preprocessors have developed. But instead of adapting FEAST's compile system to each one of them, only one preprocessor is used on all platforms.

The observation that the GNU C preprocessor `cpp` is available on all platforms FEAST has been ported to has triggered the development of `bin/f90cpp`: Usually, preprocessing and compiling is one step for the user. A compile statement invokes the compiler which in turn internally first calls the preprocessor and then tries to compile its output. FEAST always uses `cpp` to preprocess source files according to certain command line options and only then passes the preprocessed source files to the appropriate compiler. Because not all Fortran 90 compilers provide a possibility to use an arbitrary external preprocessor or the GNU C preprocessor in particular, `bin/f90cpp` has been written. All Fortran 90 source files are compiled using this script with the generic compile command

```
../../bin/f90cpp <compiler> <options>
```

The script

1. parses the command line options and filters those relevant for the preprocessor and should not be passed through to the compiler,
2. does some `sed` manipulations of the source files using regular expressions which may be difficult to read and understand for the unfamiliar user,
3. invokes the GNU C preprocessor with the appropriate command line options,
4. postprocesses the GNU C preprocessor's output with `awk` in an attempt to compensate for line number changes caused by the substitutions made by the GNU C preprocessor,
5. stores the result in a temporary file in the object directory `OBJDIR`[1] and finally
6. invokes the compiler with the (filtered) original command line options that were passed to `f90cpp`.

The actions carried out by the script are far reaching as the source code is manipulated on the fly and to some extend. Occasionally, a user may get confused because a compiler message refers to source code that differs from the one he is editing. In the coarse of this section, all operations will be explained in prose. The script is very well documented – even for FEAST standards – with 50 % of the source lines being explanatory comments and having read this section, the experienced FEAST programmer should be able to adapt and possibly extend the script.

The examination of `f90cpp` starts with a closer look at the `sed` manipulations of the original source file.

In section 5.1 on page 25 it has been explained that (non-system) header files are copied or evaluated and stored to the object directory `OBJDIR` by the Makefile. This means that path information needs to be stripped off from any reference to a header file before the file is passed to the compiler. For source files that include header files from the current directory there is no need to include path information in an `#include` (C

[1] To be more precise, directory information is obtained from the first object file or program name found in command line arguments.

preprocessor style) or `include` (Fortran 90 style) statement. But in some case, both kernel and application source files refer to header files residing in a different directory. It is necessary to include path information in the include statement. Otherwise `configure` can not determine the dependencies of those source files and will refuse to create a Makefile. But the directory information must not appear in the preprocessed source file that is passed to the compiler. Because all header files do – at compile time – reside in the object directory where the preprocessed source file is stored to. A `sed` statement employing regular expressions is used in `f90cpp` to remove the path information accordingly.

Warnings and errors in FEAST and FEATFLOW2 are raised using `error_print()`. The user might have already noticed that whenever this subroutine ends the program, the line number where and the source file from which `error_print()` has been called from is included in the message. But the corresponding `error_print()` calls do not contain this information. The magic is hidden in `f90cpp`. The C preprocessor recognises two special symbols: The symbol `__LINE__` is defined as the current line number (a decimal integer) as known by `cpp`, and `__FILE__` is defined as the current filename (a C string) as known by `cpp`. Both can be used anywhere in a source file just as any other defined symbol. All that had to be done to make file and line information of the calling point available in FEAST and FEATFLOW2 was a `sed` statement that replaces

```
call error_print(...)
```

with

```
call error_print_aux(__LINE__, __FILE__, ...)
```

a routine not referenced anywhere in FEAST, only by means of this `f90cpp` magic.

Closely related to this issue is the next: line numbers of preprocessed source files should match those of the original: The programmers edits the original source file that is passed to `f90cpp`, while the compiler acts on the preprocessed source file. Messages of the compiler and the runtime environment refer to line numbers in the preprocessed source file which do not necessarily match those of the original file. But it is very convenient for the programmer, if they would do.

If the source file contains preprocessor directives like

```
#ifdef
...
#else
...
#endif
```

source lines of the case(s) that do not evaluate to true are removed by the preprocessor and replaced with empty lines – up to a limit of 8 blank lines. If an inactive preprocessor directive block comprises more than 8 lines, *one* line is added by `cpp` that starts with a hash character (`#`) and then contains the line number in the original source file of the first statement following the preprocessor directive block. Line numbers of subsequent lines will be off by the number of lines removed. The C or C++ compiler know how to interpret these lines added by the preprocessor and use the information to to ensure that error message of the C preprocessed source code match those of the original code. Fortran compilers are not set up to interpret a GNU C preprocessor's output which is why there exists the `awk` script `bin/postprocess_cppoutput.awk`. It tries its best to compensate for line number changes by parsing `cpp`'s output instead. For a minimal module `foo.f90` like

```
  module x

    use fsystem
#ifdef ENABLE_FANCY_STABILISATION
    ! Some kind of comment.
    ! The comment does not make much sense,
    ! it's there just to demonstrate some issues.
#  define STABILISATION_PARAMETER1  1.0
```

```
  #  define STABILISATION_PARAMETER2  2.0
  #  define STABILISATION_PARAMETER3 -1.0
  #  define STABILISATION_PARAMETER4  0.3
      use state_of_art_stabilisation
  #endif
      implicit none

    !<types>
      !<typeblock>
      type x
        !...
      end type
      !</typeblock>
    !</types>
    end module
```

cpp output looks like:

```
# 1 "foo.f90"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "foo.f90"
    module x

      use fsystem
# 14 "foo.f90"
      implicit none

    !<types>
      !<typeblock>
      type x
        !...
      end type
      !</typeblock>
    !</types>
    end module
```

The syntax is explained in the leading comment of `bin/postprocess_cppoutput.awk`. If lines have been removed by `cpp`, the `awk` script can successfully compensate for this by adding dummy lines in the form of Fortran comments. If the expansion of preprocessor macros has led to additionally lines (which is the case e. g. when using the `cpp` command line option `-traditional-cpp`), there is nothing the `awk` script can do about it. Line numbers in compiler error messages and those of the runtime environment will be wrong.

Finally, `f90cpp` performs substitutions to compensate for minor annoyances: Fortran 90 source lines should not exceed 132 characters. The substitutions made by `cpp`, in particular those macros defined in Feast's `kernel/`
`feastcppmacros.h`, can easily reach this limit. But, as has been stated earlier, `cpp` operations are strictly confined to one line. To allow Fortran 90 instructions that – after the substitution – exceed 132 characters, the special keyword `MYNEWLINE` is inserted in preprocessor macro definitions. It gets replaced by a carriage return after `cpp` has processed the file.[2] Another minor issue is that the C preprocessor on NEC SX-8 disrupts the Fortran 90 pointer assignment operator `=>` as well as the boolean constants `.TRUE.` and `.FALSE.` The whitespace added by the C preprocessor is removed again with help of `sed`.

There is one drawback of using the GNU C preprocessor. `cpp` complains about every source line that contains an uneven number of single or double quotes. For the example 8 it will issue the following warning:

[2] A developer adding or updating a preprocessor macro definition in `kernel/feastcppmacros.h` should likewise insert `MYNEWLINE`'s if the expanded macro could exceed the 132 characters per line limit.

```
    foo.f90:7:11: warning: missing terminating ' character
```

because in line 7, column 11 there is a single quote without a second match on the same line. The only feasible workarounds are to either use the backtick character instead or to avoid apostrophes completely.

9 Feast and Featflow2 preprocessor macros

This feature is currently not used in FEATFLOW2 and will be documented on demand.

10 buildconf.h

FEAST decides at run time which SBBLAS settings to use. This decision is based on information stored in the configuration file `feast/feast/sbblas/sbblas.conf` and the build ID. In order to reliably determine the build ID, `fsystem` includes a header file that contains the tokens of a build ID in format keyword=”value”, one token per line. It is created, and recreated if necessary, by `bin/cc_buildconf.pl` in the course of compiling FEAST source files.

Example of `buildconf.h` as created by `bin/cc_buildconf.pl`:

```
sarch="pc"
scpu="opteron"
sos="linux64"
scompiler="intel"
sblas="goto"
smpienv="ompi"
```

11 Third party libraries

FEAST requires the third-party libraries METIS, UMFPACK and a library providing BLAS and LAPACK routines. In case any of these libraries is unavailable on your system, FEAST provides unaltered tar balls with the official reference implementations of the respective libraries. They are on demand extracted, patched if necessary and compiled. Simply set `BUILDLIB` accordingly and invoke `make`. The tar balls are expected to reside under `feast/feast/libraries` in the FEAST directory tree. If they are not, it is for licensing reasons. In this case, please download them from the internet and save the tar balls to the named directory.

To up- or downgraded these libraries, it suffices to replace the tar balls. The magic lies with the `Makefile.FEAST` for every library, stored in the respective subdirectories. Each of them is derived from the libraries' original Makefile and adapted

1. to unpack the tar ball on demand (in a parallel-safe manner, i. e. parallel `make` runs are supported, even if several applications are compiled in parallel all of which trying to compile the library),
2. to patch source files if required to work around bugs arising with certain compilers (e. g. PGI not compiling native LAPACK 3.2 sources) or to compensate for incompatible `#pragma` directives,
3. to provide a Fortran to C interface where that is required (e. g. `umf4_f77wrapper_port.c` for UMF-PACK[1]),
4. to compile only those routines that are really used in FEAST to speed up compilation,
5. to use the FEAST compile settings.

A library's `Makefile.FEAST` is automatically called by an application's `Makefile` if that library is stated in `BUILDLIB`.

11.1 Using different BLAS implementations

There are several tuned combined BLAS and LAPACK implementations available, e. g. Sun Perflib, IBM ESSL, Intel Math Kernel Library (MKL), AMD Core Math Library (ACML), MathKeisan etc. Open source alternatives like ATLAS BLAS and GOTO BLAS provide primarily BLAS routines and just a small subset of LAPACK routines. If one wants to swap one BLAS library for another, simply alter the fifth token of the . Of course, there has to a matching rule in `Makefile.cpu.inc` for this BLAS library. See section 7 on page 41 for details on how to add new rules.

[1] It has been derived from a file shipped with UMFPACK itself: `UMFPACK/Demo/umf4_f77wrapper.c`. It has been heavily adapted by Michael Köster from the FEAT 2.0 developer team and by the FEAST developers themselves. Unfortunately, interfacing Fortran and C programs is not standardised before Fortran 2003, which is why this UMFPACK interface might not be as portable and reliable as the rest of FEAST.

12 List of error messages

This section is a stub. The intension is to collect all error messages the build system may throw and to point the user to the relevant sections of this document where he finds a more elaborate explanation about what is wrong.

1. When trying to compile I get an error message like

   ```
   *** Error: Build ID foo-bar-foo-bar-foo
   ***        is *not* valid for current host!
   ***        .../bin/guess_id reported a host of type foo-bar-baz.
   ***        Cowardly refusing compilation.
   ```

   see item "confirmation that the chosen build ID matches the current architecture" on page 34.

2. When trying to compile I get an error message like

   ```
   *** Error: Invalid build ID pc-athlon64-linux-foo-bar
   ***        4th token not matched against any rule in .../Makefile.cpu.inc.
   ***        5th token not matched against any rule in .../Makefile.cpu.inc.
   ```

   The settings used to compile an application are determined by mapping the five respectively six tokens that make up a build ID to names of binaries, compiler command line flags, preprocessor directives, environment variables etc. This is done by conditionals (see page 3.1 for explanation) in , `Makefile.cpu.inc` and the Makefile templates in`Featflow2/templates/*.mk`. If there is no matching rule for a particular token in a given build ID – in the example above the tokens `foo` and `bar` are undefined – you need to add the desired settings to one of the mentioned files. Check the list of possible build IDs for the current architecture which you can query by issueing

   ```
   ./configure --list-ids
   ```

   or in case you have already generated a `GNUmakefile`

   ```
   make list-ids
   ```

   When cross-compiling the restricting of possible build IDs to those that match your current architecture is not helpful. For these cases you can list *all* possible build IDs by

   ```
   ./configure --list-all-ids
   ```

   and

   ```
   make list-all-ids
   ```

   See also item "reassurance that the current build ID is handled fully and correctly" on page 34.

3. When compiling I get many warnings of the following kind:

   ```
   foo.f90:7:11: warning: missing terminating ' character
   ```

   See for explanation and solutions the last paragraph of section 8, page 8.

4. (List to be extended. Please report any error message you get that is not yet handled here.)

Index