WCC – The WCET-Aware C Compiler

Getting Started and Documentation

Arno Luppold Dominic Oehlert Mikko Roth
Shashank Jadhav Heiko Falk Kateryna Muts

January 19, 2023

Institute of Embedded Systems
Hamburg University of Technology

Contents

1 Infrastructure

TUHH offers a dedicated server infrastructure for WCC development and usage. This chapter covers any specifics for this build infrastructure.

1.1 Available Servers

Student access is granted to the following servers:

- `essrv3.es.tu-harburg.de`
- `essrv4.es.tu-harburg.de`

Staff has additional access to:

- `essrv5.es.tu-harburg.de`
- `essrv6.es.tu-harburg.de`
- `essrv7.es.tu-harburg.de`

Additionally servers for special projects:

- `essrv8.es.tu-harburg.de`

The bug tracker of the project is available at `https://bugs.es.tu-harburg.de`. For access to the bug tracker, a separate user login is needed. Please refer to your advisor or the institute's system administrator for further details

Additionally, a Jenkins continuous integration server is performing automated build, unit and evaluation tests. It can be accessed using the same credentials for accessing the compute servers via ssh. The server can be reached via `https://essrv2.es.tu-harburg.de/`.

1.2 Access to servers

- Linux: ssh
- Windows: e.g., PuTTY
  `http://www.chiark.greenend.org.uk/~sgtatham/putty/`
- Graphically via X2Go `http://www.x2go.org`
- Supported session types in X2Go: LXDE
- KDE or Gnome do NOT work. XFCE might be buggy.

1.3 Code Repository

1.3.1 Subversion

Warning

The SVN repository is the global, shared, stable repository for WCC. If you are a student, you are *NOT* meant to commit there. Instead, refer to Section 1.3.3.

WCC's main repository is maintained using Subversion (SVN). A current version of WCC can be retreived by issuing

```
svn checkout https://svn.es.tu-harburg.de/svn/code/wcc/trunk/
```

For a nicer overview, a web frontend is available at `https://svn.es.tu-harburg.de/websvn/`.

Note

The web frontend features RSS feeds. If you go to, e.g., `https://svn.es.tu-harburg.de/websvn/wsvn/code.wcc?`, the RSS feed can be accessed by clicking on the RSS button at the right. Subscribing to the RSS feed using an rss reader (like, e.g., Mozilla thunderbird), will keep you up to date if someone commits something into SVN.

Warning

The WCC repository is organized by using SVN *externals*. This means that, e.g., ICD-LLIR is not part of the WCC repository, but an own repository which is linked into the WCC checkout path.
This means that, on submit, one must manually `cd` into the external's directory and issue an `svn commit` command there, in order to commit any changes.
Additionally, WCC's RSS feed from the web frontend will *not* show changes committed to the externals. All externals must be susbscribed on their own.
Refer to the SVN manual and/or your advisor for further information.

1.3.2 Guidelines for Submission to SVN

Anyone submitting to SVN shall obey the following rules.

1. **Think twice before committing.** Committing something to SVN has serious consequences. All other developers will get your changes once they are in SVN, and if they break something, they will break it for everybody. All commits will be available in the SVN repository forever. On the other hand, SVN allows one to revert changes, so it's possible to recover from mistakes. This is relatively easy.

   The baseline is: Be aware of the consequences of your commits. Take time to think about them before committing.

2. **Never commit code that doesn't compile.** Compile the code and correct all errors before committing. Make sure that newly added files are comitted. If they are missing, your local compile will work fine, but everybody else won't be able to compile. You certainly should make sure that the code compiles with your local setup. You should also consider what consequences your commit will have for compiling with the source directory being different from the build directory.

   Be especially careful if you change the build system, i.e. Makefiles.

3. **Test your changes before committing.** Start the application affected by your change and make sure that the changed code behaves as desired. Run regression tests if available ("gmake check").

4. **Double check your commit.** Do an "svn update" and an "svn diff" before committing. Take messages from SVN about conflicts, unknown files etc. seriously. "svn status" and "svn diff" will tell you exactly what you will be and what you will not be committing. Check if that's really what you intended.

5. **Always add descriptive log messages.** Log messages should be understandable to someone who sees only the log. They shouldn't depend on information outside the context of the commit. Try to put the log message only to those files which are really afected by the change described in the log message. In particular, put all important information which can't be seen from the diff in the log messge.

6. **The code you commit must adhere to the WCC coding policies.** Refer to Chapter 5 for further information.

7. **When you plan to make changes which affect a lot of different code in SVN, announce them to the WCC developers in advance.** Changes which affect a lot of code in SVN, like making use of a new feature in libraries, might break other code even if they look trivial, e.g. because an application must also compile with older versions of the libs for some reasons. By announcing the changes in advance, developers are prepared and can express concerns *before* something gets broken.

8. **Don't commit changes to the public API of libraries without prior review by other WCC developers.** There are certain special requirements for the public APIs of the WCC libraries, e.g. source and binary compatibility issues. Changes to the public APIs affect many other WCC developers.

9. **Take responsibility for your commits.** If your commit breaks something or has side effects on other code, take the responsibility to fix or help fix the problems.

10. **Don't commit code you don't understand.** Avoid things like "I don't know why it crashes, but when I do this, it does not crash anymore." or "I'm not completely sure it that's right, but at least if works for me." If you don't find a solution to a problem, discuss it with other developers.

11. **Don't commit code before leaving on vacation.** Avoid commits immediately before leaving on vacation. You might not be able to fix any issues created by your commit, leaving the rest of the team with the bugs you created.

12. **Don't abuse SVN to push changes with which other developers disagree.** If there are disagreements over code changes, these should be resolved by discussing them, not by forcing code on others by simple committing the changes to SVN.

13. **If you commit bugfixes, consider porting the fixes to other branches.** Use the same comment for both the original fix and the backport, that way it is easy to see which fixes have been backported already.

14. **Official WCC release and branch tags use upper case. Use lower case tags for all other tags and branches.**

15. **Don't add files generated by standard tools to the repository.** Files generated at build shouldn't be checked into the repository because this is redundant information and might cause conflicts. Only real source files should be in SVN. An exception to that are files generated by tools that would be an unusual requirement for building WCC. Standard tools include autoconf/automake, gcc, bash, perl.

16. **Make "atomic" commits.** SVN has the ability to commit more than one file at a time. Therefore, please commit all related changes in multiple files, even if they span over multiple directories at the same time in the same commit. This way, you ensure that SVN stays in a compileable state before and after the commit.

17. **Don't mix formatting changes with code changes.** Changing formatting like indenting or whitespaces blows up the diff, so that it is hard to find code changes if they are mixed with reindenting commits or similar things when looking at the logs and diffs later. Commiting formatting changes separately solves this problem.

Warning

Ignoring these rules may have severe consequences for both others and yourself. It make break code of other developers, and can severly disturb their work, including progress on papers or their thesis.

1.3.3 Mercurial (HG)

In order to provide a safer playground for local changes, including the possibility to commit completely broken work, collaborate with other WCC developers, branch, fork, etc. without any risk of interfering with the official SVN repository, a Mercurial (`hg`) repository is provided. This repository is completely file-based and resides on the institute's servers in a shared directory.

- Version control with Mercurial (Introduction: https://www.mercurial-scm.org/wiki/)

- Repository for students: `/es_groups/wcc/hg/wcc-studi`
- Repository for staff members: `/es_groups/wcc/hg/wcc-staff`
- Repository clone: `hg clone /es_groups/wcc/hg/wcc-studi localDirectory`

Warning

Do *NEVER EVER* commit *ANYTHING* into the "default" branch. It is solely there to reflect the current state of the Subversion repository. Consider the "default" branch as *read only*.
If you commit to "default" accidentally, do *NOT* push your changes. Instead, notify your advisor. He or she will help you.

- Use `hg branch <Projectname>` to create an own branch for your work.
- *Do* regularly perform `hg merge -r default` to merge current changes from the default-branch into your own branch.
- `hg pull` gets the current changes from the common repository.
- `hg push` pushes the locally committed changes into the common repository.

Note

`hg` may fail at pushing new branches. Read `hg`'s error message to find out how to proceed in this case.

Warning

The idea behind version control is to manage source code and to keep an overview of the changes done. This means: Do *NOT* commit or push any files which have been created during the build process of the WCC.
Especially do *NOT* blindly `hg add` all files of the readily built WCC into the repository! In case of doubt of problems related to Mercurial please consult your advisor first.

1.3.4 Tips for Mercurial (not a complete overview)

The file ~/.hgrc should be edited to contain your name and email address. Furthermore additional features of Mercurial can be activated here:

```
[ui]
username = FirstName LastName <firstname.lastname@tuhh.de>
merge = meld
[extensions]
purge=
```

In addition, the `prj-wcc` group might have to be added as "trusted" to the ~/.hgrc. Otherwise, pushing change sets to the shared repositories may fail due to missing trust of the .hgrc file in the remote repository. This can be achieved by adding the following to the local ~/.hgrc:

```
[trusted]
groups = prj−wcc
```

hg revert <file>	Revert changes of a file.
hg revert -r <rev> <file>	A specific previous revision of a file can be restored.
hg up -C	Undo all changes of all files which are part of the version control.
hg purge	Delete all files which are not kept under version control requires the entry purge= in the file ~/.hgrc).
hg strip -r <rev>	Removes revision "rev" from the local (!) repository. If a "hg push" has already been executed, "hg strip" does not have any effect anymore, since the locally removed revision will be pulled again when "hg pull" is executed the next time...

Table 1.1: Frequently used mercurial commands

1.4 The Module System

In order to serve specific user demands on the server, specific software can loaded via the module system. The following commands are useful when working with the WCC:

module avail	Shows all available modules.
module load <name>	Loads a module.
module unload <name>	Unloads a module.
module list	Lists all currently loaded modules.

For building the WCC you require the wcc meta-module, which loads all the other required modules. This should happen automatically when you login to the servers. Check the loaded module with the command:

```
module list
```

If you get the "No Modulefiles Currently Loaded.", try the following command:

```
module load wcc
```

You can modify the modules which are automatically loaded (or unloaded) at login by editing the file `~/.modules.linux`.

Listing 1.1: Exemplary .modules.linux config file

```
+wcc
+cbmc
-libboost-ia32/1.60.0-gcc5.4
```

This file automatically loads the `wcc` and `cbmc` modules at login. Additionally, the module `libboost-ia32` in version `1.60.0-gcc5.4` is *un*loaded. If this module had not been loaded in exactly this version, nothing will happen. File file is parsed from top to bottom.

1.5 Tips for SCREEN

Some evaluations or build processes may take a long time. The tool *screen* offers a comfortable way to keep a process running on a remote server, even when the remote connection fails or beyond logout.

screen (`https://www.gnu.org/software/screen/`) is started by typing `screen` in the terminal. The interface of screen is very rudimentary by default which makes it somewhat hard to use. Therefore, it is recommended to adjust the configuration of screen via the file ~/.screenrc:

```
1  # change the hardstatus settings to give a window list at the
2  # bottom of the screen, with the time and date and with
3  # the current window highlighted
4
5  hardstatus alwayslastline
6
7  hardstatus string '\%\{= kW\}\%-Lw\%\{= kG\}\%50> \%n\%f* \%t\%\{= kW\}\%+Lw\%<
       \%\{=kW\}\%-= \%S - \%H\%\{-\}'
8
9  #read and write screen clipboard to X clipboard.
10 bind > eval writebuf "exec sh -c 'xsel -nbi </tmp/screen-exchange'"
11 bind < eval "exec sh -c 'xsel -bo >/tmp/screen-exchange'" readbuf
```

Attention: The cryptic term behind `hardstatus string` has to be placed in one single line! With this addition, screen now shows a status bar in the footer with additional information.

- With `CTRL+a` the command mode is activated (which is not visually displayed).
- With `CTRL+a,:sessionname foo` the screen session is renamed to "foo".
- With `CTRL+a,CTRL+d` the screen session is deattached. All processes keep running, but the current session is not shown anymore in the terminal.
- Deattached sessions can be shown again via `screen -r`
- Via `CNTRL+a,c` you can create a new tab inside the current screen session.

1.6 Tips for VS Code

Visual Studio Code is a popular code editor. To program on the WCC code base using VS Code follow the tips in this section.

1.6.1 Installation

VS Code is not installed per default on the PC pools and servers. To install it for your user account follow these steps:

1. Visit `https://code.visualstudio.com/download` and download the latest `64 bit` version for Linux in the `.tar.gz` format.
2. Extract the contents of the archive to a preferred location inside your home directory. (On the command line: `tar -xzf <filename>`)
3. Now you can launch VS Code by executing the file `./VSCode-linux-x64/bin/code`

1.6.2 Remote Development

In case you want to work from your personal computer, you may also install VS Code on your local machine and connect to the servers remotely.

1. To connect via SSH to the servers, install the `Remote - SSH` extension ($View \rightarrow Extensions$ (`Ctrl+Shift+X`)).
2. Add the server as a remote host by executing the command `Remote-SSH: Add new SSH Host` (`Ctrl+Shift+P`).
3. Input your account name and desired server (e.g.`name@essrv3.es.tuhh.de`)
4. Connect to the server: `Remote-SSH: Connect to Host`. (You have to accept the fingerprint of the server on the first connection.)
5. Now you can edit files remotely and execute commands on the server ($Terminal \rightarrow New\ Terminal$).

2 Getting Started

2.1 Compiling WCC

For building WCC, we first start by calling the following command line option in the terminal within the WCC's repository. It creates the build system for WCC (`https://en.wikipedia.org/wiki/GNU_Build_System`). Secondly we create a build directory within the WCC's repository so that we can keep the automatically generated files apart from the source files. Third, we move to the build directory and finally use `./configure` to set WCC's compilation options.:

```
1 make –f Makefile.svn
2 mkdir ./build
3 cd build
4 ../configure [OPTION]... [VAR=VALUE]...
```

To assign environment variables, for example, CC, CFLAGS, etc, we can specify them as `VAR=VALUE`. To see all the options that `../configure` provides we can use `--help` option. The most frequently used configuration options are mentioned below.

`--enable-debug` Enables debug system

`--with-doxygen-doc=no` Do not create documentation (default=yes)

`--with-arm-support` Build for ARM target architecture (default=no)

`--with-leon-support` Build for Leon target architecture (default=no)

`--with-release-quality=yes` Build with GCC's `-O3` optimization level (default=yes). Do not use this for debugging but only for faster evaluations.

`--with-licensee=NAME` Give NAME as licensee. Needed in combination with release quality being enabled.

If WCC is configured without using `--with-arm-support` or `--with-leon-support`, WCC is built by default for the TriCore target architecture. The output of the debug messages is controlled by WCC's configuration switch `--enable-debug` or else, the debug outputs are omitted strictly. Next, to compile WCC use the following command:

```
make –j[N]
```

This allows running N jobs in parallel at once. To see all the options that `make` provides we can use `--help` option.

2.2 Compiling Programs with WCC

To compile a C source code file with WCC from the source code directory we use the following command line option (within the build directory):

```
./WCC/wcc/wcc [OPTIONS]... [FILE]...}\
```

This calls WCC for compiling C source code file `[FILE]`. By using `--help` option we can explore all the available options for optimizations and analysis within WCC. Following some common WCC options for compilation.

`-vX` Set verbosity level to n ($0 \leq X \leq 9$, default = 0), see Table 2.1

`-v` Display the programs invoked by the compiler (same as `-v1`)

`-o <file>` Place the output file into `<file>`

`-On` Set optimization level to n ($0 \leq n \leq 3$, default = 1)

`-Owcet` Compute the WCET

The verbose message that the user receives from `-vn` provides some more useful information about the current status and progress of an optimization or analysis. It depends on the value of `'n'`, which is the verbosity level, that how chatty the WCC should be. The following information is received for the different level of verbosity:

The output of the log files is controlled by WCC's `--keepX` class of command line options. If specified on the command line, log files with arbitrary content can be produced in the current working directory. Following are some of the usually used `--keepX` options (all these options are off by default).

`--keepTemp` Keep compiler intermediate files.

`--keepIR` Keep ICD-C IR before code selection in file test.ir.

`--keepLLIR` Keep ICD-C LLIR from code selection

`--KeepBackannInfo` Keep back-annotation status file. This options should be used in conjunction with `-Owcet`.

2.3 Doxygen Documentation

- A current version of this document can be found in the sub-folder DOC.
- In the sub-folder DOC you can find documentation on the supported architectures (TriCore/ARM7) as well as guidelines regarding coding styles and similar.
- The program code of WCC has been documented doxygen compatible.

Table 2.1: WCC verbosity levels

-v0	No verbose messages are emitted at all.
-v1	One line of information about the currently performed optimization PLUS the command line of invoked system calls.
-v2	`-v1 PLUS` Summarized effect of the individual optimizations using e.g. LLIR_Statistics PLUS the global WCET of the final optimized program if `-Owcet` is given.
-v3	`-v2 PLUS` CPU runtimes of each individual optimization.
-v4	`-v3 PLUS` Detailed results for each individual optimization.
-v5	`-v4 PLUS`Detailed progress report per optimization or analysis, i.e., one line per sub-step of an optimization (e.g. Constructing interference graph, Precoloring interference graph, Inserting spill code, ...)
-v6	`same as -v5`. Verbosity level 6 is reserved for future use. It must not be used by anybody.
-v7	`-v6 PLUS` Output of the system calls issued by libllirait.
-v8	`-v7 PLUS` Output of the code selector
-v9	`-v8 PLUS` Output of the used register allocator.

- This means, that by setting the ./configure parameter `-with-doxygen-doc=yes|no` the automated cration of HTML documentation on the classes and functions in WCC can be triggered.
- After building WCC by issuing make, you can find the documentation in the sub-directory `doc` under the respective top-level directory in WCC.
- Exemplary, the documentation for LLIR-level optimizations can be found at `LLIROPTIMIZATIONS/doc/html/index.html`

2.4 Handling Problems and Asking Questions

For questions related to content or organization please refer to your advisor.

If you have access to WCC's bug tracker, please file your bug under `https://bugs.es.tu-harburg.de/`

For question related to the WCC, *any* given error, program behavior, a (possible) bug, please *always* include (at least) the following information. Even if you think some or all of these information is not necessary, please *do* provide them.

- Subversion or Mercurial-Revision (`hg summary`). Make sure, that there are no changes which have not been committed yet and that the current revision has been pushed into the common repository via `hg push`.

- Complete `./configure` parameters, with which the WCC has been built.
- Complete calling parameters of the WCC, with which the behavior can be triggered.
- Detailed description of *all* steps necessary in order to reproduce the behavior.
- Exactly describe the (faulty) behavior of the compiler, as well as the expected correct behavior.
- Please do not attach huge log files to mails. Instead, describe the steps necessary in order to reproduce the error. Often the devil is in an (seemingly unimportant) detail.
- Question like "I execute the WCC and get some weird error" can in general not be answered meaningful due to missing background information.

Warning

If you are working on WCC in pursuit of a student thesis, you are *STRONGLY* encouraged to clean up your code in accordance to *ALL* rules and recommendations presented in Chapter 5 *PRIOR* to asking your supervisor.
Please do *NOT* ask any questions why your code does not work as expected when working on a completely messed up code base. It is your duty to make your code comprehensible – not that of your supervisor.
Ensure that running `hg diff -r default` on your development branch produces a *CLEAN* and *well-arranged* diff. If this is not the case, then do clean up your code prior to asking questions.

3 Advanced Usage

This chapter covers some more or less advanced topics on how to call and use WCC.

3.1 Command Line

WCC's main features can be steered by using command line options. The parameter `--help` gives an overview of commonly needed options. `--help -v` gives a much more complete overview.

3.2 Configuration Files

Apart from its command line parameters, multiple settings can be configured by a `wccrc` config file (e.g., the used target-gcc compiler, memory limits for aiT, preferred aiT version, target CPU frequency,...

For further information, check out the `wccrc` for the respective architecture. It is found under WCC *build* directory in the sub-directory `WCC/wcc/etc/ARCH/`.

Here, also the memory specification file (ARCHlayout.mem) can be found.

Note

Multiple locations are searched for the `wccrc` by WCC. The order is:

- `etc/ARCH/wccrc` under WCC's installation directory (or: build directory if make install was not issued)
- `$HOME/.wccrc`
- `$PWD/.wccrc`

Be aware, that parsing does *NOT* stop after finding the first config file. Instead, all files are parsed.
It is option specific (and may be even configured, depending on the option), what happens if the same option is encountered multiple times in differend wccrc config files. Compare, e.g., CONTEXT_LIMIT_WRITE_MODE. For further information, ask somebody with deeper knowledge or check out the source code starting at processOptionFiles() in `wccoptions.cc`.

3.3 Multi-Task

Functions can be annotated as entrypoints in the source code. The most basic annotation looks like this.

Listing 3.1: Basic Entrypoint Marker

```
// Entrypoint function.
void _Pragma( "entrypoint" ) foo() {
  ...
}
```

This marks the function foo() as a task. Multiple tasks may be annotated within each compilation unit as needed.

Note

In order to correctly link the program, it will still need a main() function. Yet, main() does not have to be marked as an entrypoint.
As soon, as at least one function is explicitly marked as an entrypoint, main() will *not* be considered as an entrypoint, unless also explicitly marked by a _Pragma directive.

The entrypoint marking above will mark the function as an entrypoint only. This is sufficient to perform a WCET analysis for each entrypoint.

However, the entrypoint does not yet have any timing requirements like, e.g., deadline or period, assigned to it. Subsequently *schedulability* analysis is not yet possible within WCC.

The following listing shows an entrypoint with timing requirement annotations:

Listing 3.2: Entrypoint Marker with Timign Requirements

```
// Entrypoint function.
void _Pragma( "entrypoint period=10000 jitter=100 deadline=20000 priority=1" ) foo() {
  ...
}
```

This assigns period, jitter, deadline and a fixed priority to the task. Timings are given in clock cycles.

3.4 System Specification Files

TODO: *Describe* `.tasks` *files* TODO

3.5 Multi-Core

TODO: *Add some specifics on how to model/define/... multi-core setups, buses,...* TODO

3.6 Automatic Setting Generation for Evaluation

For evaluating optimizations or analyses, it may be useful to automatically generate system properties like, e.g., periods, deadlines, SPM size,....

For this, WCC offers the command line parameter `-fEvaluate`. This should not be used when solving real-world problems.

Instead, its main purpose is to ease usage of WCETBENCH benchmarks and evaluate a given optimization or analysis for different system parameters. I.e., an SPM optimization may be easily run this way with different SPM sizes without having to tamper with any configuration files.

data-section Define the section where data objects (i.e., everything which would go to `.data` by default) will be allocated to.

text-section Define the section where instruction (i.e., everything which would go to `.text` by default) will be allocated to.

Example: `wcc -fEvaluate --param text-section=".text_cached"` will allocate all code to the section named `.text_cached` .

Warning

Allocating data and text will only work for code which is handled by WCC. I.e., code and data generated by WCC from source files.
Code and data by external libraries (e.g., floating point library, startup files,...) which are only linked at the very end will *not* be affected by this.

dspm-size Modify the size of the data SPM. This may be given in an absolut number (in byte) or as a percentage.

pspm-size Modify the size of the instruction SPM. This may be given in an absolut number (in byte) or as a percentage.

Note

Percentage values are relative to the size of data (instructions, respectively) of the program which is compiled.
As for the section allocation described above, external libraries are not accounted for.

Example: `wcc -fEvaluate --param pspm-size=40% -o prime.elf prime.c` This will resize the SPM to be 40 % the size of the code of the prime.c benchmark. I.e., at most 40 % of the prime benchmark may be allocated to the SPM by an optimization.

Note

In practice, due to the fact size of the basic blocks and the resulting knapsack problems, a 100 % utilization of the SPM will probably not be possible.
Also, note that an SPM size of 100 % might not suffice to move the whole program into SPM. This stems from the fact, that moving all basic blocks to the SPM may lead to the situation that calls to external functions must be re-written as indirect calls due to larger displacements (because the external function now resides in a different memory region). As a result, basic block sizes increase, and 100 % are no longer 100 %...

load Specify a target load. This can be used to automatically generate periods for the entrypoints in the system using UUniFast algorithm.

jitter Specify a random jitter

lowerDeadline, upperDeadline Specify bounds of the deadline.

Example:

Listing 3.3: Call of WCC with -fEvalute

```
wcc -fEvaluate --param load=1.2 --param jitter=0.02
  --param lowerDeadline=0.8 --param upperDeadline=1.2 -o file.elf source.c
```

For this example, first a WCET analysis is performed. Based on the WCETs of all entrypoints defined in `source.c`, periods are calculated such that the overall load of the task set is at 1.2 (i.e., the system will *not* be schedulable in this example). Additionally, a jitter of 2 % is defined. This means, that uniformly randomly a value between 0 and $0.02 \cdot$ `period` will be calculated which acts as activation jitter. `lowerDeadline` and `upperDeadline` will randomly dice a deadline in the range `lowerDeadline` $\cdot$ `period`, `upperDeadline` $\cdot$ `period`.

3.7 Annotating Flowfacts to Libraries (ARM)

When compiling libraries for ARM7TDMI (`-c` parameter), flow facts are annotated into the object file using dwarf code.

This can also done manually by calling

`ICD-LLIR-ARM/arch/tools/objff <object file>`

Type `help` at the appearing prompt for some cool help parameters.

TODO: *Apart from that, does* anybody *know how to work with this cool stuff? Then please, go ahead and add documentation ...* TODO

4 Debugging WCC

Before proceeding with debugging, ensure that your code passes the quality control checks which are presented in Section 5.2. Also ensure your code complies with all coding style rules presented in Chapter 5. Often, cleaning up "messy" code already suffices to find and/or fix an issue. In any case, it will significantly ease any subsequent debugging steps.

Warning

If you are working on WCC in pursuit of a student thesis, you are *STRONGLY* encouraged to clean up your code in accordance to *ALL* rules and recommendations presented in Chapter 5 *PRIOR* to asking your supervisor.

4.1 Compilation Options

In order to allow for efficient debugging, `--enable-debug` must be passed to `./configure` when building WCC. Otherwise, the debug output system described in the following section will not work.

4.2 Working with Debug Macros

If WCC has been built with debug mode enabled (see previous section), multiple macros may be used to ease debugging. The next subsection will first describe the most common macros. Then, usage is described in a seaprate section.

4.2.1 Macro Description

The following example shows a function with multiple debugging macros.

Listing 4.1: Example for Using Debug Macros

```
#include <config_wcc.h>

#include "libuseful/debugmacros.h"

using namespace std;

int MyClass::foo()
{
  DSTART( "MyClass::foo()" );
```

```
  int a = 5;
  DOUT( "The value of a is: " << a << endl );

  DACTION(
   ++a;
   );

  DOUT( "The new value of a is: " << a << endl );

  return a;
}
```

It is mandatory to include config_wcc.h in order for the debug macros to work. Otherwise, the debug macros will not notice whether debugging was enabled or not as part of `./configure` stage. The default is to *not* enable debugging.

Debug output can be enabled or disabled on function level at run-time without having to re-compile WCC. Functions are identified by an arbitrary string which is supplied to DSTART( ) at the head of a function. WCC's coding style suggests to use the class and function name as identifier.

Apart from DSTART(), the most common macros are DOUT() and DACTION(). DOUT() can be used as a debugging version of cout. DOUT() output, however, is only enabled if debugging for this function was enabled. Therefore, there is no need to remove debugging output once a class is finished. Instead, it can be left for future uses.

The DACTION() macro can contain one or multiple C++ statements. It can be used for more sophisticated code which should only be executed if this function is actively being debugged. In the example above, the variable a is only incremented, if debugging is enabled for this function. Be aware that this means, that, in the example, the return value of the function is dependent on debugging being active or not. Situations like this should be *avoided* by any means.

Sometimes, the DDECLARE() macro may come handy. It can be used to declare a variable only in case of debugging. This avoids "unused variable" warnings on debugging variables when compiling with release quality.

The macro DINIT() can be used to parse an external config file. In this config file, the actively debugged functions can be defined (see next subsection). If a custom DINIT() is needed, please refer to an existing usage in WCC (e.g., in `fsm.cc`).

Warning

When compiling with `--with-release-quality`, all debug macros are removed. Therefore, make sure that debug macros do not have *any* side effect. I.e., avoid situations like in the example above where ++a; increments a only if debugging is enabled and thus the return value of the function changes, depending on debugging being on or off.

4.2.2 Activating Debug Macros

Debugging macros are enabled by config files which are usually placed in the `DEBUGMACRO_CONF` directory in WCC's source tree.

For some classes, config files are pre-defined but disabled (file suffix `.bak`). Remove the `.bak` suffix to enable them.

It is *not* mandatory, that each class has its own config file. It is only mandatory that the config file is parsed by a DINIT() macro prior to calling the function which is to be debugged.

Note

Usually, `fsm.conf` may be used to enable debugging on any class in WCC. Only very few classes are executed prior to the FSM.

The format of the configuration files is very simple. Each line contains the string which was defined in the DSTART() macro of the function to be debugged. For the example from the prvious section, the configuration file would look like this:

Listing 4.2: Exemplary debug config file

```
MyClass::foo()
```

When running WCC, this will then trigger output each time foo() is entered. Additionally, a line will be written to stdout denoting if foo() is exited. Additional to this, all DOUT() in the function generate output to stdout and the code in the DACTION() clauses will be executed. WCC idents the function entry and exit writeout by DSTART() as well as all DOUT() outputs. This way, recursive functions, as well as debug output of multiple functions which call each other can easily be distinguished.

4.3 GDB - The GNU Debugger

If WCC was compiled with debugging enabled, the GNU debugger can be used to efficiently debug WCC.

Especially if WCC crashes with a core dump being generated, it is very easy to find the position in the code, the call chain, and probably a hint on why WCC crashed at that position using `gdb`.

To generate the core dump:

1. If you work on your own laptop and not on the server, you might need to set first

   ```
   ulimit -c unlimited
   ```

2. Run WCC

   ```
   <BUILDDIR>/WCC/wcc/wcc <parameters> <inputCFiles>
   ```

   After this the core dump should be generated in the current directory.

3. Call `gdb`

```
gdb −c <generatedCore> <BUILDDIR>/WCC/wcc/.libs/wcc
```

To see the call chain, use `gdb`'s command

```
bt
```

Refer to any guide on `gdb` and google if you don't know how to handle `gdb`.

Warning

Please do *NOT* ask your supervisor on how to use `gdb` without having read, tested and understood at least some tutorials. It might also be helpful to get accustomed to `gdb` by trying it out using a small custom minimal standalone example, first. Then, once you've learned how to operate `gdb`, you can try it on WCC.
Finally, note, that there are some nice GUI tools which might help you using the advanced fatures of `gdb`.
On the institute's current infrastructure, `geany`, `qtcreator` or `kdevelop` could prove to be useful.

4.4 Memory-Related Issues

There are basically two ways which ease debugging memory related issues. Both approaches may be used individually or may also be combined:

- Compile-time parameters
- External tools at run-time of WCC

4.4.1 Compile-Time Parameters

For this, stack protection and the address sanitizer can be used. These are two parameters that can be passed to g++ when compiling WCC.

-fstack-protector-all This enables stack protection. This is primarily not meant as a debugging feature but as a security feature. However, it can be very helpful at debugging, since it will cause the application to crash when bad things happen on the stack - and not at some arbitrary point somewhere later in the program's execution.

-fsanitize=address This is a very sophisticated debugging tool originally introduced by LLVM and nowadays also available in GNU g++. It basically replaces malloc system calls and is specifically tailored towards detecting and debugging use after free, heap buffer overflows, stack buffer overflows,... See `https://github.com/google/sanitizers/wiki/AddressSanitizer` for more information.

Both parameters can automatically be activated during WCC's build process by using configure parameters. `--enable-memory-debugging` enables both stack protection and address sanitizer. Alternatively, `--enable-memory-protection` enables ONLY the stack protector compilation flag. As an alternative, the paramters could also be passed by using the well-known CFLAGS and CXXFLAGS environment variables.

Note

Combine `--enable-memory-debugging` with `--enable-debug`. I.e., a configure call might be: `./configure --with-doxygen-doc=no --enable-debug --enable-memory-debugging`
Depending on your local Ubuntu system and installed debugging symbols for system libraries, an additional `--enable-x64` might also be useful.

Note:

- Address sanitizer and memory protection are very powerful tools to detect memory-related issues. In most cases, simply compiling with this should find the source of memory-related crashes without any further actions needed.
- Enabling the address sanitizer will drastically slow down both WCC's runtime performance and the time needed to build WCC. Building WCC may easily take 2-3 times longer than without address sanitizer and WCC's runtime may also easily drop by a factor of 2-3.
- Address sanitizer will most likely also find issues which are not related to the specific bug (e.g., auto-generated code in CODESEL and code in misc/anyoption.cc). Anyone's welcome to fix these issues. :-)
- You might have problems building WCC successfully due to failures in the final build stage where WCC is called in order to build floating point libraries (fp-bit-s.o) and the like. If this happens:
  - Build WCC in a separate directory and NOT in the source dir (This is crucial for the subsequent steps!)
  - Copy the target libraries over form another build, e.g.: rm -rf WCC/wcc/lib cp -r /working/build/WCC/wcc/lib WCC/wcc/lib
  - Now call WCC the way that caused the original crash and have fun debugging. :)

4.4.2 External Tools

Apart (or in addition to) using the address sanitizer and stack protector, `valgrind` is the tool to be used when it comes to detecting memory leaks/bugs. `valgrind` must be called on the WCC binary itself, *not* on the libtool wrapper script which is put to WCC/wcc/wcc after compilation. I.e., to call valgrind:

- Build WCC (best with debugging enabled). Depending on your local system you might have to compile WCC with `--enable-x64`.
- Open WCC/wcc/wcc in a text editor and find the line setting the LD_LIBRARY_PATH environment variable
- Change to your terminal. Copy&paste the LD_LIBRARY_PATH there, such that it is defined in your local shell.
- Now call: valgrind `--leak-check=full $BUILDDIR/WCC/wcc/.libs/wcc <WCC parameters as needed>`

Notes:

- You might not have sufficient permissions to run valgrind properly. Do *NOT* run valgrind as root. Instead, read and follow the output of valgrind. It will tell you what to do (most likely setting some value to "0" or "-1" somewhere under /proc/) If you don't have root access, due to working on the institute's servers, please refer to your advisor and/or the institute's system administrator.
- Valgrind will heavily slow down WCC's runtime performance. You might wantto grab a coffee. item In order to debug memory leaks, you can get additional information by additionally passing `--track-origins=yes` to valgrind. However, this will drastically decrease runtime performance even further.
- You can do cool stuff and have valgrind check at run-time if some variable is initialized by adding check points into the source code like this:

Listing 4.3: Valgrind Code Instrumentation

```
#include <valgrind/memcheck.h>
...
// Check, whether *so is defined and print if it is uninitialized
VALGRIND_CHECK_VALUE_IS_DEFINED( *so );
```

Note

Valgrind will always find a significant amount of memory leaks in ICD–C, auto-generated CODESEL code, as well as in LLIR_LoopNestingForest (and probably some more). You can either ignore or fix them. :-) However, they will most likely be unrelated to your special bug/crash/issue.

4.5 Performance Bottlenecks

TODO: *perf, callgrind* TODO

5 Coding Guidelines

This chapter gives an overview of important things for *WCC developers*. It defines WCC's coding style, covers mandatory code quality checks as well as I/O verbosity levels.

5.1 Coding Style

5.1.1 Formatting

- No Tabs
- Indent with 2 spaces
- No trailing white spaces
- A line must not have more than 80 chars
- If the arguments of a function don't fit into the 80 chars, start a new line and align the additional arguments with the first one.
- Put spaces between operators and arguments
- Put spaces between brackets and arguments of functions
- For pointer and reference variable declarations put a space between the type and the $*$ or & and no space before the variable name.
- For if, else, while and similar statements put the brackets on the same line as the statement
- Function and class definitions have their brackets on separate lines
- Each C++ source code file must end with the extension .cc
- Each C++ source code file must start with a comment as shown in Listing 5.1

Listing 5.1: Formatting Example

```
/*

   This source file belongs to the

            Hamburg University of Technology (TUHH)
               WCC Compiler framework

   and is property of its respective copyright holder. It must neither be used
   nor published even in parts without explicit written permission.

   Copyright 2007 - 2022

   Hamburg University of Technology (TUHH)
   Institute of Embedded Systems
   21071 Hamburg
   Germany

   http://www.tuhh.de/es/esd/research/wcc

*/


void MyClass::myFunction( MyPointer *pointer, MyReference &ref,
                          Argl *oerx )
{
  if ( blah == fasel ) {
    blubbVariable = arglValue;
  } else {
    blubbVariable = oerxValue;
  }
}
```

5.1.2 Header Formatting

- General formatting rules apply
- Access modifiers are indented
- Put curly brackets of class definitions on its own line
- Double inclusion protection defines are all upper case letters and are composed of the namespace (if available), the classname and a H suffix separated by underscores
- Inside a namespace, there is no indentation
- Each C++ header file must start with a comment as shown in Listing 5.2

Listing 5.2: Header Example

```
/*

   This header file belongs to the

           Hamburg University of Technology (TUHH)
              WCC Compiler framework

   and is property of its respective copyright holder. It must neither be used
   nor published even in parts without explicit written permission.

   Copyright 2007 - 2022

   Hamburg University of Technology (TUHH)
   Institute of Embedded Systems
   21071 Hamburg
   Germany

   http://www.tuhh.de/es/esd/research/wcc

*/


#ifndef _XKJ_MYCLASS_H
#define _XKJ_MYCLASS_H

namespace XKJ {

//! This is the short description for class MyClass.
/*!
   This is the more detailed description for class MyClass.
*/
class MyClass
{
  public:
    //! Default constructor.
    MyClass();


  private:
    int mMyInt;
};

}

#endif // _XKJ_MYCLASS_H
```

5.1.3 Header Conventions

- Any header should start with the include guard / double inclusion protection. There should be nothing especially not any **#include** directives before the include

guard.

- Headers should only include headers of classes whose definition they need. E.g. if your header only uses a pointer or a reference to an object of type T, then do not include the header for type T, but use a forward declaration of T instead to reduce the amount of dependencies between headers. This saves time during compilation. On the other hand, if T is used as a data member in your header or if T is a template, you will need to include the header for T.
- Never put a **using namespace** std; in header files. Namespaces should be kept clean. Thus, directives like **using namespace** std; should only occur in C++ source code files. In header files, always use a qualifier std:: when using standard data types like e.g. std::string, std::list, etc.

5.1.4 API Docs

- Each class declaration must be preceeded by a single-line concise comment starting with *//!* in order to achieve doxygen compatibility.
- Whenever useful, more detailed class descriptions can be added after the single-line short description using the doxygen */∗!* and *∗/* comment delimiters.
- See the above code snippet for an example of a valid class documentation.
- Each public function must have a concise single-line comment starting with *//!* in the header.
- Whenever useful, more detailed public function descriptions can be added after the single-line short description using the doxygen */∗!* and *∗/* comment delimiters.
- See Listing 5.3 for an example of a valid public function documentation.
- Comments should be grammatically correct, e.g. sentences start with uppercase letters and end with a full stop.
- Be concise.
- Be sure to document all paramters, the returned data and possibly thrown exceptions. Especially make sure that you document the beaviour in case of an error. In case of boolean return values, precisely state when true and when false is returned.
- Make sure that your code does not generate any doxygen warnings when WCC is built with documentation.

5.1.5 Class and File Names

- Put classes in files, which have the same name as the class, but only lower-case letters

Listing 5.3: Function Documentation Example

```
//! This function makes tea.
/*!
    More specifically, it makes 'cups' cups of tea.
    @param cups The number of cups which should be made
    @return A pointer to the cup of tea which was made. The caller must
    ensure that the object is delete-d when not needed anymore.
    @throws my_custom_error if no teapot could be found.
*/
Tea *makeTea( int cups );
```

5.1.6 Class and Variable Names

- For class, variable, function names separate multiple words by upper-casing the words preceeded by other words
- Class names start with an upper-case letter
- Function names start with a lower-case letter
- Variable names start with a lower-case letter
- Member variables of a class start with "m" followed by an upper-case letter

5.1.7 Misc

- Use **true** and **false** instead of TRUE and FALSE. These are the C++ keywords after all.
- A null pointer is nullptr, not 0, 0l, 0L or NULL. Once again, this is C++11, not C++98 or C.
- Program const-correct. Declare Parameters as const if they are not modified in your function. Declare your function const if it does not modify any member variables of your class. Declare your function static, if it does not access any member variables.
- Use const references instead of call by value for non-trivial parameters. I.e. **void** my_fun(**const** std::string &name); instead of **void** my_fun(std::string name).
- Use references instead of pointers where possible.
- Use ufAssert() and ufAssertT() to verify any assumptions on input and output parameters.
- Make sure your code does not generate any warnings with g++, doxygen and cppcheck static code analyzer.

Note

Some older parts of WCC may not adhere with these guidelines. Especially, they may still use 0 or NULL instead of nullptr for pointer initialization. If such cases are found, please update the code according to these style guidelines.

5.1.8 Error Handling

This section solely tackles handling of errors. For Debug and progress output refer to the appropriate section below. We distinguish between assertions and 3 classes of errors: Class-Internal Errors which can be recovered from within the current class, non-fatal errors which cannot be recovered from within the current class and fatal errors.

Assertions Assertions shall only be used for error conditions which are considered to be impossible in a non-buggy implementation. If an an assertion is raised, this is always considered to be a bug in WCC. Regular error conditions shall be indicated by throwing an appropriate exception (see below), instead.

- assert(cond): assert() defined by the "cassert" header should be avoided if possible. It may be used for assertions which will not be present when compiling WCC with `--with-release-quality` configure switch enabled. This can be handy for very time consuminga ssertions or in highly timing critical functions. However, be aware that in a "release-quality" build, these assertions will not be present.
- ufAssert(condition): Whenever possible, ufAssert() should be used to assert requisites. See Listing 5.4 for an example

  Warning

  Do *NOT* use something like ufAssert(condition && "Error message"); to print error messages. This will not only lead to ugly output, but also cause warnings in the `cppcheck` static code analyzer. Always use ufAssertT() if a message should be printed in case the assertion is violated.

Listing 5.4: Assertion Usage Example

```
// Include LIBUSEFUL headers
#include <libuseful/io.h>

ufAssert( condition );
ufAssertT( condition, "Meaningful error Message" );
```

Function-Internal Errors If an error occurs within a class's function which can be recovered from within this function itself, an error message of type ufErrMsg must

be output by the function observing the error before operation is resumed. Check section 5.1.9 for further information. Cf. Listing 5.5 for an example.

Listing 5.5: Class-Internal Recoverable Error Example

```
// Include IO headers
#include <libuseful/io.h>

// ...
ufErrMsg << ufFile() << "Error message" << endl;

// Resume normal operation
```

Recoverable Errors If an error within a class's function leads to the abnormal termination of that function but does not cause any side effects, i.e. WCC's overall runtime consistency is guaranteed, an exception of type ufError shall be thrown:

Listing 5.6: Recoverable Error Example

```
// Include exception headers
#include <libuseful/exceptions.h>

// ...
throw ufError( "A meaningful error message" );
```

Note

Exceptions of class ufError may be catched and handled by a calling class.

Fatal Errors If an error may lead to undefined behavior of WCC, e.g., due to a possibly inconsistent LLIR, or if the error might prevent correct compilation due to any other reason, a ufFatalError exception shall be thrown. ufFatalError should be considered unrecoverable and shall not be catched by any class other than WCC's main() routine.

Listing 5.7: Fatal Error Example

```
// Include exception headers
#include <libuseful/exceptions.h>

// ...
throw ufFatalError( "A meaningful error message" );
```

Custom exceptions Custom exception classes may be created for libraries, optimizations,... Custom exceptions shall inherit publicly from either ufError or ufFatalError. It must be ensured that calling what() on the exception prints a meaningful error message.

Warning

Do *NOT* call exit(), abort() or similar in a library or optimization in case of an error. These functions will directly terminate without calling the destructors of any local or dynamically allocated objects. Always throw a ufFatalError or a custom exception which inherits from ufFatalError in case of a non-recoverable error!

5.1.9 Verbosity, Logging and Debugging Output

We distinguish five different classes of output, a WCC module can make: Error messages, warnings, verbose messages, debug output and log files. For these classes, the following rules apply (strictly):

Errors The output of errors can never be suppressed. Error messages have to be generated using LIBUSEFUL's IO library:

Listing 5.8: LIBUSEFUL Output Example

```
// Include IO headers
#include <libuseful/io.h>

// A warning message
ufWarnMsg << ufFile() << "Warning message" << endl;

// An Error message
ufErrMsg << ufFile() << "Error message" << endl;

// Fatal error message
ufFatalMsg << ufFile() << "Fatal error mesage" << endl;
```

Refer to LIBUSEFUL doxygen documentations for more usage information.

Warnings The output of warnings is controlled by WCC's -Wx class of command line options. If specified on the command line, warning messages are emitted using LIBUSEFUL's ufWarnMsg.

Verbose Messages Verbose messages may provide the WCC user with some more useful information about the current status and progress of an optimization or analysis module. The output of verbose messages is controlled by WCC's -vx class of command line options where the user can specify a verbosity level, i.e. a degree of how chatty WCC should be. Use WCC's IO library by including misc/wccio.h.

Listing 5.9: WCCIO Output Example

```
// Include IO headers
#include <misc/wccio.h>

// Message printed for verbosity level 1 and higher
WCCIO::v1Msg << ufFile() << "Verbosity 1 message" << endl;
```

```
// Message printed for verbosity level 2 and higher
WCCIO::v2Msg << ufFile() << "Verbosity 2 message" << endl;

// ...

// Message printed for verbosity level 9 and higher
WCCIO::v9Msg << ufFile() << "Verbosity 9 message" << endl;
```

Refer to MISC doxygen documentations for more usage information. However, the output of verbose messages must adhere to the following rules:

Table 5.1: WCC Verbosity Rules

Verbosity Level	**Action**
0	No verbose messages are emitted at all.
1	One line of information about the currently performed optimization PLUS the command line of invoked system calls.
2	-v1 PLUS summarized effect of the individual optimizations using e.g. LLIR_Statistics PLUS the global WCET of the final optimized program if -Owcet is given.
3	-v2 PLUS CPU run-times of each individual optimization.
4	-v3 PLUS detailed results for each individual optimization. For WCET-aware optimizations, this MUST include a program's WCET before and after the optimization. Other relevant information may e.g. be the partition size per task etc.
5	-v4 PLUS detailed progress report per optimization or analysis, i.e. one line per sub-step of an optimization (e.g. "Constructing interference graph", "Precoloring interference graph", "Inserting spill code",...).
6	-v5 PLUS Verbosity level 6 is reserved for future use. It must not be used by anybody.
7	-v6 PLUS output of the system calls issued by libllirait.
8	-v7 PLUS output of the code selector.
9	-v8 PLUS output of the used register allocator.

Debug Output The output of debug messages is controlled by WCC's configure switch `--enable-debug`. Only if `--enable-debug` is given, debug outputs are allowed. Otherwise, they have to be omitted strictly. Debug output has to be generated using libuseful's debug macros DINIT(), DSTART(), DOUT() and friends.

Log Files The output of log files is controlled by WCC's `--keepx` class of command line options. If specified on the command line, log files with arbitrary content can be

produced in the current working directory. However, if no `--keep` option is given, no log files are produced at all.

5.1.10 Directory Structures

- Each module of WCC is placed in a distinct subdirectory under the top-level WCC directory. Make sure that configure.in and Makefile.am are adjusted when adding a new WCC module.
- In a module's subdirectory, the following files have to be maintained: `AUTHORS`, `ChangeLog`, `COPYING` and `Releases`. Optionally, files like e.g. `ACKNOWLEDGMENTS` can be added.
- Each module of WCC needs to be provided by means of a (static or shared) library. The source codes of the library are placed in a subdirectory like e.g. `<WCC top-level>/REGALLOC/regalloc`
- The library needs to provide a function named string <the library's name>::Version() returning the current version number of the WCC module. This version number is extracted from the module's `Release` file at make time and is compiled in the library's code using preprocessor definitions (see e.g. REGALLOC for this mechanism).
- Besides the subdirectory containing the library source codes, other subdirectories may exist. For example, `<WCC top-level>/<WCC module>/tests` should contain the module's testbench, or `<WCC top-level>/<WCC module>/tools` some additional binaries for the module.

5.2 Quality Control while Coding

Compiler Warnings Your source code should never produce compiler warnings.

cppcheck The tool *cppcheck* is available on the servers and is able to find additional errors or points of slackness with the help of static code analysis. It can be applied on your own code via:

```
module load cppcheck
cppcheck --language=c++ --enable=all -j4 *.cc *.h
```

The man page (man cppcheck) shows additional information. Existing code in WCC might produce cppcheck warnings. You are welcome to eliminate existing errors. Make sure, that the fix does *not* have any bad side effects, though.

5.3 Jenkins Server

The test server of the WCC can be reached under `https://essrv2.es.tu-harburg.de/`. This server builds the WCC in different configurations and checks it for errors.

Using the job `wcc-validate-patch` you can test your own code by uploading the changes in comparison to the default branch as a patch file. This patch file can be generated via:

```
hg diff –r default > patch.diff
```

You can find more information using `man patch` or `man diff` or inside the documentation of Mercurial.

5.3.1 wcc-evaluation-opt

This section describes the Jenkins's job *wcc-evaluation-opt*, which runs WCC's optimizations. The job runs optimizations for **TriCore** listed in the file <SRCDIR>/WCC/WCETBENCH/evaluation/lists/configurations/**testopt-tc**.list and optimizations for **ARM** listed in the file <SRCDIR>/WCC/WCETBENCH/evaluation/lists/configurations/**testopt-arm**.list.

The job uses the benchmarks from <SRCDIR>/WCC/WCETBENCH/evaluation/lists/templates/testserver.list.

To **add a new optimization** (see, e.g., <SRCDIR>/WCC/WCETBENCH/evaluation/configurations/wcetopt/fWCET-ilp):

1. create a new directory for the optimization <OPTDIR> in the corresponding directory of <SRCDIR>/WCC/WCETBENCH/evaluation/configurations, e.g, in wcetopt;
2. create files **wccflags** and **excludes** in <OPTDIR>;
3. add optimization flags to the file **wccflags**
4. add benchmarks to be excluded from evaluations for this new optimization in the file **excludes**
5. add the optimization to
   - <SRCDIR>/WCC/WCETBENCH/evaluation/lists/configurations/**testopt-tc**.list,
   - <SRCDIR>/WCC/WCETBENCH/evaluation/lists/configurations/**testopt-arm**.list
   - or both.

To test the job before pushing the changes to SVN: in WCC's source directory (<SRCDIR>), execute the following command

```
ACTION=evaluate–opt . wcctest.sh
```

6 Performing Evaluations

6.1 WCETBENCH: Evaluate Optimizations and Analysis

First Build WCC in the build directory, and then ...

```
1 cd WCC/WCETBENCH/evaluation
2 ./new-run.sh --help
3 ./new-run.sh --arch <arch> --objectives wcet <SRCDIR>/WCC/WCETBENCH/evaluation/
      lists/configurations/<optimizations>.list <SRCDIR>/WCC/WCETBENCH/evaluation/lists/
      templates/<BenchmarkSuite>.list
4 cd runs/<createdConfiguration>
5 nice make -j16
```

- **IMPORTANT:**
- Always adapt the priority via `nice` to prevent disturbing other users.
- Do not use a TOO HIGH parallelism.
- Start `nice make ...` in a `screen` (see Ch. 1.5)
- This way, evaluations which take a long time are not aborted when you sign off.

7 Adding Functionality

7.1 Adding Optimization/Analysis Modules

To add a new optimization or analysis to the WCC the following files have to be modified:

1. /MISC/misc/wccoptions.cc
2. /MISC/misc/configuration.h
3. /FSM/fsm/optimizer.h

For optimizations targeting specific architectures the following files need to be modified as well:

1. /LLIROPTIMIZATIONS/llirARMopt/llirARMopt.cc
2. /LLIROPTIMIZATIONS/llirTCopt/llirTCopt.cc
3. /LLIROPTIMIZATIONS/llirLEONopt/llirLEONopt.cc

wccoptions.cc: Add the command line switch for your module. This switch adds the optimization/analysis to the list of module which will be called during compilation with the WCC.

```
./WCC/wcc/wcc  ... -<SWITCH>
```

The following example adds an analysis for the ARM to the WCC:

```
if ( theArg == "-tandem-flow-analysis" ) {
config.extendOptimizationSequence( Configuration::TANDEMFLOW );
} else
if( theArg == "-fno-tandem-flow-analysis" ) {
config.removeFromOptimizationSequence( Configuration::TANDEMFLOW );
} else
```

The first one adds the module to the optimization sequence of the WCC. The WCC will execute the module if the switch "`-tandem-flow`" is set on the command line or in the configuration file. The second instruction removes the analysis from the list of scheduled analyses.

configuration.h : Introduce the analysis to the WCCs configuration class. The configuration is generated and initialized during initialization of the WCC and contains the information which analyses and optimizations will be run:

```
TANDEMFLOW,         // Congestion-aware analysis
```

optimizer.cc : Add the optimization to the optimizer object which is called by the FSM to start the optimizations:

```
LLIR_Optimizer llirOptimizer( mSystem, mConfig, mMemLayout);
if( mConfig.getOptimizationFlag( Configuration::TANDEMFLOW ) )
stat += llirOptimizer.mtOptimize( tc, Configuration::TANDEMFLOW );
```

llirARMopt.cc : Add the analysis to the specific `llir*opt.cc` of the target architecture (ARM, LEON, TriCore). Here the analysis will be called during compilation with WCC. For the ARM case we add the following to the `llirARMopt.cc`:

```
case Configuration::TANDEMFLOW: {
mConfig.printVerboseMessage( VERBOSE_1,
"Performing tandem flow analysis." );
LLIR_TandemFlow tandemFlow;
tandemFlow.analysis();
break;
}
```

8 Multi-Core Framework

WCC features support for multi-core architectures. On such an architecture, code can be compiled for each core in one run of WCC. This is useful for multi-core aware WCET analysis and optimizations.

At the time of writing this document, WCC has support of multi-core systems featuring compilation units which are based on ARM7TDMI. For these systems, WCC features its own internal bus-aware WCET analysis framework.

8.1 Introduction

This section gives a brief overview of the class design of a multi-core system in WCC.

1. The class TaskConfig holds the configuration and global memroy layout for the system. Basically, TaskConfig comprises the whole system with all source files, assembly files, configurations,...

2. A TaskConfig contains one or multiple objects of type TaskEntry. Each TaskEntry describes one computational unit, i.e., one *core*. item Each TaskEntry contains an IR which holds the C code to be compiled, a list with all LLIRs, which hold the corresponding low-level representation, a Configuration object which holds WCC's configuration settings for this core, and a LLIR_MemoryLayout which holds the memory layout for this core.

Note

Be aware, that both TaskConfig and TaskEntry feature pointers to a Configuration and LLIR_MemoryLayout objects. In case of a single-core setup, those may have the same content or even point to the identical object. In case of multi-core setups, both the Configuration, and LLIR_MemoryLayout between the cores and between the cores (i.e., the TaskEntry) and the system (i.e., the TaskConfig) may differ.
Be sure to *always* refer to the correct configuration and memory layout objects, depending on your intentions.

Warning

Despite its name, a TaskEntry describes a *processing unit* (i.e., a *core*) and *NOT* a task.
TaskConfig is the global object class which comprises the full system and *NEITHER* a "configuration" *NOR* a single task or its behavior.
Tasks which are executed on a core are called *entrypoint* and modeled by the class LLIR_Entrypoint. Refer to Chapter 9 for further information.

Multi-Core systems are usually described using XML-based `.tasks` files. These configuration files can be passed to WCC on the comand line similar to a *C* file in traditional compilation. The XML defines all cores of the system and specifies which C files should be compiled for each core.

8.2 WCET-Analysis

TODO: *Internal bus-aware multi-core WCET analysis* TODO

8.3 Event-Arrival Framework

TODO: *Documentation of LIBRTC Framework* TODO

9 Multi-Tasking Framework

Chapter 8 introduced the class design within WCC to describe a system and its computational cores.

Within each core, multiple *entrypoints* may be defined. These can either be defined as annotations within the *C* source code, or as part of WCC's XML-based `.tasks` files.

Marking a function as an entrypoint specifies that this function acts as an independent task. I.e., it can be called by some scheduler process or interrupt. When performing WCET analysis, the WCET analysis is called multiple times, with each of these functions being assumed as the *entry* of the program.

Entrypoints are managed by the IR_Entrypoint class on ICD-C level and by the LLIR_Entrypoint class on LLIR level.

In addition to the plain entrypoint marker, entrypoints can also be assigned activation patterns (e.g., a fixed period and a jitter) a deadline and a fixed priority (in case of fixed-priority scheduling). Due to the fact that these additions are realtively new, they are not integrated into the ICD-C and LLIR entrypoint concepts but simply added as string pragmas to the source- and assembly-level code constructs.

Custom multi-task classes and helper functions, as well as the LIBMULTIOPT framework are used to handle these tasks.

Note

In compliance with literature, WCC defines deadlines as unsigned integer numbers with numerically lower numbers signifying a logically higher. I.e., the task with the priority 0 has the highest priority in the system.
This is convenient, as by this way, a task's deadline may be directly used as its priority in deadline-monotonic scheduling. Respectively, its period may be directly used as priority in rate-monotonic scheduling.

For usage information, refer to Chapter 3.

9.1 LIBMULTIOPT Framework

The underlying algorithms for multi-tasking analysis and optimization reside under LIBMULTIOPT directory.

This library is independent from WCC but only relies on LIBUSEFUL and LIBILP and provides multiple functionalities. The standalone character allows for easy creation of unit tests verifying the correct behavior of the classes.

Non-complete list for basic task handling:

RTAEvent Purely virtual interface class which defines the API for event function handling.

RTATask Base class for modeling taks (i.e., entrypoints in WCC nomenclature).

RTATaskPJ Models periodical tasks with jitter.

RTATaskMPJ Models tasks which are triggered by multiple periods with jitter.

RTATaskset Combines multiple tasks into a task set.

RTASchedTests Provides schedulabiltiy tests for both EDF and fixed-priority preemptive scheduling.

SysGenerator Helper-Class for timing property creation. This provides an implementation of UUnifast algorithm, as well as an algorithm to calculate the hyper period of a periodic task set and an algorithm to adapt tasks' periods such that the hyperperiod is reduced.

The class also features basic routines for optimizations. The functionality is with a focus on WCC but in order to being able to use unit testing and verify the class' behavior more easily, this has been implemented in a WCC-independent way.

RTASensitivity Sensitivity analysis which returns an estimate on how many (and which) tasks must be removed from a task set in order to make it schedulable.

RTASensitivityOpt Same as RTASensitivity, but based on ILP.

RTASensitivityOpt The functions provided in this class add ILP constraints and variables which enforce schedulability. This is used by WCETILP_Multitask to provide the basis for schedulability-aware ILP optimizations (cf. LLIR_OptimizeILPProgramSPM.

LIBMULTIOPT also features basic support for incrementally growing ILPs using an oracle approach. This provides the possibility to start optimizing a system with only a small and subsequently easy to solve ILP. The user-provided oracle decides whether the solution of the incomplete ILP already suffices the design demands. If so, the problem is solved. If not, the oracle has to provide a new constraint which invalidates the previously proposed solution. This is continued until either the problem is solved, or no solution is found. In the latter case, the oracle may be changed to a more sophisticated oracle (as long as the new oracle's reply does not not contradict any of the first oracle's replies). This allows that multiple schedulability-aware optimizations are chained if necessery in order to produce a schedulable system. Refer to the doxygen documentation of the class Oracle for further information. Also check out the classes Solver, SolverEllipsoid and SolverILP. For minimal examples, check the unit tests.

Warning

This is a proof of concept only. It has not yet been implemented into a full-fledged optimization in WCC. It might need some tweaking before usage.

9.2 WCC Support

Target independent scheduling analysis in WCC is implemented in LLIROPTIMIZATIONS/lliranalysis/scheduling. The class uses the algorithms in LIBMULTIOPT to provide scheduling and sensitivity analysis.

The scheduling algorithm is taken from the Configuration. If necessary and enabled, a UCB/ECB CRPD analysis exists in order to predict cace-related preemption delays. (cf. LLIR_AnalyzeCRPD.

If target-specific changes are needed, these can be added under llirARCHanalysis/scheduling/...

Note

For schedulabiltiy analysis, a WCET is needed for each entrypoint. This can be retrieved from both WCC's internal WCET analyzer and aiT.

10 Integration of static WCET Analyzer aiT

10.1 Introduction

WCC performs WCET-aware compilation and optimization. To achieve this, WCC is tightly coupled with AbsInt's WCET analyzer aiT so that the WCET of a single-task application is estimated during compilation. Due to the tight integration of the WCET analyzer with WCC, this analysis is done fully automatically once the user calls WCC with the corresponding flags so that a C program is compiled and a WCET analysis is carried out at compile time. Once the WCET data is computed by aiT, it is imported into WCC's back-end by attaching it to the compiler's Low-Level IR. The following WCET-related data is made available within WCC this way:

- Global WCET of the entire program, WCET of each function and each basic block,
- worst-case call frequency per function,
- worst-case execution frequency per basic block,
- worst-case execution frequency per *control flow graph* (CFG) edge,
- execution feasibility of each CFG edge,
- cache hits and misses.

WCC supports Infineon TriCore's TC1796 and TC1797, ARM's ARM7TDMI and Cortex-M0, and Gaisler's Leon3 as target architectures.

10.2 Integration of aiT

The license manager of aiT's latest revision 20.10, which serves as a unified and common WCET analysis infrastructure, preventing the direct interaction with aiT's analysis tools. Instead, one dedicated program called `alauncher` is responsible for all interactions with aiT and its license management, and the interfaces provided by `alauncher` solely base on the XTC format.

alauncher takes an XTC file as one of the inputs and then executes the analyses defined therein without showing or starting a graphical interface. Another input file required by aiT to run analyses is an annotation file that contains details about the target architecture configuration as well as user-provided annotations like, e.g., flow

facts or memory access patterns. Later, the results of the analyses are found in the XML output files specified in the XTC file that is then parsed by WCC to extract analysis results and WCET data from them.

Based on the target architecture for which compilation is being done and the compilation flags set by the user, both the XTC and the annotation files are auto-generated by the compiler during runtime. Besides the links to the executable to be analyzed and to the location of the annotation file, the XTC also specifies options for value analysis, stack analysis, etc., hardware options like, e.g., stack address, stack area, etc., LP solver options, cache options, analysis entry-point information, etc.

To generate the inputs for aiT, WCC's memory layout specification plays a key role. In particular, memory regions containing the stack, regions with read or write accesses, memory region properties, address ranges of memory accesses, etc., are determined and passed to aiT. Thus, AbsInt's latest version 20.10 of the aiT WCET analyzer is completely encapsulated in WCC. The compiler user is unaware of the fact that timing analysis is performed in the background due to the smooth integration of aiT with WCC. The user does not get in touch with the configuration of parameters mandatory to run a static WCET analysis, which makes WCC user-friendly in the sense that the user does not have to face the burden of setting up a valid run-time environment for aiT.

This whole new WCET analysis framework of WCC is activated by invoking WCC with the command line argument `-Owcet`. This switch affects that, after having applied all high- and low-level optimizations, WCC performs a WCET analysis of the final, optimized binary executable program. WCET analysis results are placed in two files called `wcet.log` and `report.txt`. The first one contains a summary of analysis results, e.g., the program's global WCET, or the WCETs, and instruction cache misses per function. The latter one contains detailed logs produced by each of aiT's analysis steps and includes (amongst other information) analysis warnings or errors. Using WCC's command line option `-v`*n* with *n* being an integer between 0 and 9, the user can control the amount of information displayed by the compiler.

10.2.1 Steps to integrate the new version of aiT within WCC

To integrate a new version of aiT within WCC, we need to focus on the following repositories: MISC, AITINTEGRATION, and FLOWFACTS. It is recommended to only keep two aiT versions active to maintain the code base clean. Therefore, while integrating the new version of aiT, we can refer to the already existing code for the older version of aiT. Unless there is an extreme change in the handling of the license manager, change in the interface between WCC and aiT (which is highly unlikely), change in the format of the XML output file from aiT, or change in the AIS2 annotations used by aiT.

1. Firstly, in `configuration.h` (in MISC/misc/) we need to add new enum class for the new version of aiT in the `enum` class **AiTVersion**.

2. Within `wccoptions.cc` (in MISC/misc/), we need to add the code to determine the loaded module and set the new version of aiT for analysis. It is done for all

the architecture. Furthermore, we need to set paths to aiT executables for the new version of aiT. This is done by setting paths to **alauncher** for each of the architecture.

3. **AITIntegration** class in `aitintegration.cc` (in AITINTEGRATION/aitintegration/) acts as a framework to perform analyses using the different versions of aiT. This class consists of two functions `performAitAnalysis` and `computeELFWCET` manages the entire analysis without and with elf file as input file, respectively. These functions call a specific version of aiT that is currently loaded. Therefore, we need to add code here for the new version of aiT, so that the interface between WCC and the new version of aiT is set.

4. We need to further add four new files for the new version of aiT. For example, if the new version of aiT currently added is 20.10 then add `ait2010.cc, ait2010.h, aiti02010.cc` and `aitio2010.h`. The code in these files can be replicated by using the already implemented code for the older versions of aiT. `aitio2010.cc/.h` files consists of details for communucation with aiT 20.10 toolchain. For example, `enums` for **alauncher** and for numerous intermediate files generated along.

5. `ait2010.c/.h` files consist of details to perform analyses, parse the XML output file, process all the contexts, process the aiT analyses paths, insert WCET/energy edges to LLIR at the basic block, function, and LLIR level, and add WCET/energy objectives to LLIR. Furthermore, this file consists of code to generate the APX file and AIS files that act as an input file for the **alauncher**. These details should mostly carry forward from the older version of aiT to the newer version of aiT as it is.

6. While generating an APX file, it is necessary to check the details needed for this input file according to the new version of aiT. For example, the project, build, version, target, etc nodes within the file are according to the new version (c.f. emitapxfile(..) function within `ait2010.cc` file). To check these details, you can generate a new dummy project within aiT GUI, save the project and check the APX file generated. Also, you can check the aiT manual for this new version.

7. The AIS file which we generate consists of a lot of important information like the details to add memory specification, external/indirect call/jump/return information, loop bound and flowfact information, memory access information, etc. The AIS specification can change or get updated from version to version. Therefore, it is very important to check the AIS annotations chapter for the aiT manual and update the AIS annotations generation within WCC. Currently, for the aiT 2010 version, we are following AIS2 annotations.

8. To add memory specification and memory accesses, we use the information from the memory layout files for different architectures which are within WCC/wcc/etc/(architecture folder)/.To add jump/call/return information to the AIS file, we traverse through the LLIR to gather this information (please refer to

the existing code to find the details). To add loop bound and flowfact information, we use the Flowfact mechanism that is integrated within WCC. Check `flowfactmanagerllirtoais.cc` file (in FLOWFACTS/flowfacts/) where is information is extracted and added to the AIS file. If the AIS annotations are changed in any way, these are the places where we need to incorporate appropriate changes.

9. It is important to check if the parsing of the XML output file is correctly done. There are assertions correctly inserted in places to avoid any unforeseen mistake. We use `boost` property tree to perform parsing of the XML file. Therefore, we need to evaluate the XML file from the new version and see if any of the XML edges and the property tree hierarchy is changed.

10. Once the XML file is parsed we create equivalent LLIR edges to the given XML edges, i.e., return, loopback, loopexit, call, etc., LLIR edges. The WCET/energy values for edges are calculated, and then we create WCET/energy objects for basic blocks, functions, and LLIRs and attach the calculated WCET/energy values to these objects appropriately. Possibly, the attached WCET/energy values to LLIR can be different than the analysis values provided by aiT for the new version of aiT. In that case, the edges are not parsed properly, or the WCET/energy objects are not attached properly to LLIR/function/basic block. So, in that case, we need to check the XML file generated by the new version of aiT and the edge generation for this newer version of aiT.

11. Furthermore, AITINTEGRATION repository consists of different files, like `aitgenericfunctions.cc/.h, aitintegration.cc/.h, llir_ait_objective.cc/.h, llir_edge.cc/.h,` and `llirwcet_obj.cc/.h`, which consists of common code base necessary for aiT analyses. PROFILING repository consists of the `llirenergy_obj.cc/.h` files, which consists of LLIR energy objectives. For the integration of the new version of aiT within WCC, we most probably do not need to make any changes.

Note

To perform WCET analysis using AbsInt's aiT, use `-Owcet` command-line option. To perform Energy analysis using AbsInt's EnergyAnalyzer, use `-OenergyAit` command-line option.

Warning

Currently, AbsInt's EnergyAnalyer supports energy analysis just for Cortex-M0.

Note

The above subsection describes the steps to integrate a new version of aiT. But, it does not include the steps to integrate the WCET/energy analysis using aiT for a new architecture. If you want to add WCET/energy analysis for a new architecture PLEASE refer to the whole WCC code base :-)

11 Energy Analysis Framework

11.1 General Information

WCC's energy analysis framework estimates the average-case energy consumption of a system using the following information:

- The Average-Case Execution Count (ACEC) of each basic block
- The Average-Case Call count of each function ("call frequency")
- Execution rates (e.g., period) of each entrypoint in a multi-tasking system.
- An energy specification in XML which describes the energy consumption of a given target architecture.

The first stage of the energy analysis framework first determines the energy consumption of each LLIR_Instruction of each LLIR_BB of each LLIR_Function of each LLIR of each TaskEntry contained in the global TaskConfig. For these LLIR_Instructions, the energy consumption of one single execution of the instruction is determined. In order to allow for a more precise analysis, each instruction is analyzed in the context of a given preceeding LLIR_Instruction. This way, inter-instruction costs can be modeled. This cost is *not* aware on how many times the instruction is actually executed at program runtime (or if it is executed at all). These costs are defined by the user in the energy specification written in XML (cf. Section 11.3).

The second stage of the energy analysis estimates the execution counts of each basic block in the context of each LLIR_Entrypoint. Typically, this may be the *average*-case execution counts of each basic block. Depending on the algorithm behind this analysis, execution counts may be floats. I.e., it is totally valid that a basic block is executed 1.7 times on average in a "typical" execution of a given LLIR_Entrypoint. This allows for profiling based average-case execution counts. The different execution count estimators are discussed in Section 11.4.

11.2 Usage

This section gives a general overview on how to call and use the energy framework.

Note

Please always refer to WCC's command line help (`--help -v`) for latest usage information.

The energy analysis framework is activated by passing `-OEnergy` on WCC's command line.

Note

At the time writing this text, energy specification is only available for the thumb instruction set for the ARM7TDMI target. I.e.: Compile WCC with ARM support and additionally supply `-mthumb` on the command line.

There are currently the following ways for ACEC and function call frequeny estimation:

- A probabilistic ACEC Estimator
- Profiling using CoMET simulator
- Energy Estimation based on Worst-Case Execution Count information provided by aiT's WCET analysis

The estimator to use can be chosen by using a `--param` command line option when calling WCC:

The command line argument `--param acecEstim=heuristic` will activate the statistical ACEC estimation (default).

The command line argument `--param acecEstim=wcec` will activate the WCET-analysis based ACEC estimation. Be aware, that this may lead to strange results. All basic blocks which are *not* on the worst-case execution path at the time of the WCET analysis will have a worst-case execution count (WCEC) of 0. As a result, a subsequent optimization might wrongly assume these basic blocks are unreachable and will not optimize them. WCEC-based energy analysis may be used in some scenarios in order to provide some worst-case-ish energy consumption for architectures where energy consumption between different instructions does not differ significantly.

The command line argument `--param acecEstim=sim` will activate the platform simulator CoMET as basis for ACEC estimation.

Warning

Currently, CoMET is *not* able to simulate programs using the ARM7TDMI thumb instruction set. Thus, for ARM7TDMI with thumb instruction set, does not support simulation based ACEC estimation.

11.3 Energy Specification File (Usage)

The energy analysis uses an XML file which describes the energy behavior of the system to analyze. This file typically resides under `WCC/wcc/etc/<arch>/energy.xml`. The path to the file can be changed by setting ENERGY_DESCRIPTION in wccrc accordingly. The path to the default energy.xml is specified in the `wccrc` fil under `WCC/wcc/etc/<arch>`.

There are three main regions within the XML file: <general></general> is meant for *general information*, e.g., target architecture,... <instructions></instructions> specifies the energy consumption of *instructions* and <memory></memory which specifies the energy consumption of *memory accesses*.

Within LLIR_Operation passes both of these blocks. Each block consists of a number of possible nested <**case**></**case**> blocks. For all these blocks, a first-deepest-match approach is applied. I.e., the energy analyzer traverses from from top to bottom of the XML specification. If a <**case**> matches, it tries to traverse into optionally available sub-cases. This allows to model some operation highly detailed while modeling other operations very coarse-grained at the same time, as needed by the system designer or the target platform's behavior.

A case can either just provide logical matching, or it can contribute to the operations energy consumption in any way. An optional parameter <cumulative>**true**|**false**</cumulative> in each case can be used to specify, whether the match in case should overwrite previously accounted energy consumption for this operation, or add up to it.

If no case matches at all for an operation, energy analysis will fail with an error.

Warning

There's no need that the deepest case must match. I.e., if only the outer case environment matches in a nested case structure, this will not produce an error. use the FAIL EnergyModel to trigger an error if this behavior is not desired.

Below, find a small, artificial, example for a simple instruction block:

Listing 11.1: Small example of an XML specification

```
<instructions>

  <case> <ID>OPCODE</ID> <opcode>ADD_16</opcode>
    <energy>10.0</energy>

    <case> <ID>PARAMS</ID>
      <number>3</number>
      <energy>4.5</energy>
    </case>
  </case>

  <case> <ID>FALLBACK</ID>
    <energy>4.2</energy>
  </case>
</instructions>
```

This snippet will match an operation with the mnemonic ADD_16. For each of these mnemonics, 10.0 energy are added. The actual energy unit does not matter to WCC. Some unit may be defined for usability in the general section, but as long as a common unit is used for all cases, this does not matter for energy estimation.

The nested case PARAMS is an EnergyModel which checks the number of parameters of the operation. I.e., if the ADD operation has only 2 parametrs, it will not match,

and an energy consumption of 10.0 is returned. However, if the operation has 3 parameters,additional 4.5 are added and 14.5 is returned.

The FALLBACK case will match *everything*. Therefor, for any other instruction, an energy consumption of 4.2 will be returned.

The next sections give an overview of the currently available energy models for the XML specification.

Warning

This list may be outdated. Always refer to the doxygen documentation and source code to find a current list of all energy models with their valid parameters

11.3.1 BRISTOL

The BRISTOL model can be used in order to use the linear energy model as provided by the University of Bristol. The syntax is:

Listing 11.2: BRISTOL EnergyModel

```
<case> <ID>BRISTOL</ID>
      <prefetch>true|false</prefetch>
      <waitstate>true|false</waitstate>
      <freq>24000000</freq> <!-- CPU Frequency in Hz -->
      <coefficient>8.511066029660545e-10</coefficient>
      <intercept>0.0031227347891140678</intercept>
      <mse>0.010816947731421573</mse> <!-- optional default 0.0 -->
      <rms>0.10400455630125813</rms> <!-- optional, default 0.0 -->
      <r2>0.9972740103219596</r2> <!-- optional, default 0.0 -->
      <cumulative>true|false</cumulative> <!-- optional, default true -->
</case>
```

11.3.2 COREID

The COREID model is a purely logical case block. It matches for a user-definable Core ID. This may be useful in multi-core enviroments with big-little CPUs, where the energy consumption of machine operations differs depending on the core on which they are exeucted.

Listing 11.3: COREID EnergyModel

```
<case><ID></ID>
  <core>0</core> <!-- integer ID of the core to match -->
</case>
```

11.3.3 FAIL

The FAIL model is a purely logical case block. It will alway throw a ufFatalError exception, leading to the termination of WCC with a user-definable error message. This can be used as a safety measure to mark unreachable blocks in the XML specification.

Listing 11.4: FAIL EnergyModel

```
<case><ID>FAIL</ID>
  <msg>Optional Error Message</msg> <!-- Optional error message. -->
</case>
```

11.3.4 FALLBACK

The FALLBACK model will *always* match. It can be used as a fallback to match operations which do not have to be modeled explicitly.

Listing 11.5: FALLBACK EnergyModel

```
<case><ID>FALLBACK</ID>
  <energy>0.0</energy> <!-- Energy consumption which should be added -->
  <cumulative>true|false</cumulative> <!-- optional parameter, defaults to true -->
</case>
```

11.3.5 HAMMING

Account for the hamming distance of an operation to its predecessor. The hamming distance is the number of bits in the operation which are flipped. E.g., for an 8 bit architecture, if the preceding operation has the binary represenation 00001010 and the operation under test has 10001000, the hamming distance will be 2, as two bits differ. This model will multiply the number of differing bits with a user-definable factor.

Warning

This model has to be specialized for each target architecture, as the correct binary representation of each instruction must be known to the model.

Listing 11.6: HAMMING EnergyModel

```
<case><ID>HAMMING</ID>
  <factor>0.8</factor> <!-- User-definable factor -->
  <cumulative>true|false</cumulative> <!-- optional parameter, defaults to true -->
</case>
```

11.3.6 MULTOPS

This model matches several mnemonics at once. It can be used to cluster multiple operations which feature the same energy consumption characteristics.

Listing 11.7: MULTOPS EnergyModel

```
<case><ID>MULTOPS</ID>
  <energy>10.0</energy> <!-- Energy to add -->
  <opcode>ADD_16</opcode> <!-- Mnemonic to match -->
  <opcode>SUB_16</opcode> <!-- another mnemonic to match -->
```

```
  ...
  <cumulative>true|false</cumulative> <!-- optional parameter, defaults to true -->
</case>
```

At least one opcode block must be present. The mnemonic corresponds to the one defined in ARCH/proc.h

11.3.7 OPCODE

This is similar to the MULTOPS model, but only supports one single mnemonic.

Listing 11.8: OPCODE EnergyModel

```
<case><ID>OPCODE</ID>
  <energy>10.0</energy> <!-- Energy to add -->
  <opcode>ADD_16</opcode> <!-- Mnemonic to match -->
  <cumulative>true|false</cumulative> <!-- optional parameter, defaults to true -->
</case>
```

11.3.8 OFFSET

This model can be used to model whether a given parameter of an operation is an offset (e.g, an immediate). It does not model the mnemonic of the operation itself, but must be nested inside an OPCODE or MULTOPS case if it should only match a certain operation.

Listing 11.9: OFFSET EnergyModel

```
<case><ID>OFFSET</ID>
  <position>1</position> <!-- The position of the parameter, starting at 0 -->
  <energy>10.0</energy> <!-- Energy to add -->
  <cumulative>true|false</cumulative> <!-- optional parameter, defaults to true -->
</case>
```

11.3.9 PARAMS

This model matches, if the operation under test has the user-defined number of parameters.

Listing 11.10: PARAMS EnergyModel

```
<case><ID>PARAMS</ID>
  <number>2</number> <!-- Number of parameters the operation must have to match -->
  <energy>10.0</energy> <!-- Energy to add -->
  <cumulative>true|false</cumulative> <!-- optional parameter, defaults to true -->
</case>
```

11.3.10 PREDECESSOR

This model matches, if the *predecessor* of the operation under test has a given mnemonic. Note, that this does not traverse basic block boundaries.

Listing 11.11: PREDECESSOR EnergyModel

```
<case><ID>PREDECESSOR</ID>
  <opcode>ADD_16</opcode> <!-- The mnemonic to match, as defined in proc.h -->
  <energy>10.0</energy> <!-- Energy to add -->
  <cumulative>true|false</cumulative> <!-- optional parameter, defaults to true -->
</case>
```

11.3.11 REGISTER

this model matches, if the parameter at a given position is a user-definable register.

Listing 11.12: REGISTER EnergyModel

```
<case><ID>REGISTER</ID>
  <position>2</position> <!-- The parameter which is tested, starting at 0 -->
  <register>R4</register> <!-- The register to match (string comparison) -->
  <energy>10.0</energy> <!-- Energy to add -->
  <cumulative>true|false</cumulative> <!-- optional parameter, defaults to true -->
</case>
```

For the **register** block, three special registers exist: ANY will match any register. This can be used to distinguish a parameter from, e.g., an OFFSET. SP resolves to the stack-pointer register, as defined in proc.h PC resolves to the program counter, as defined in proc.h

11.3.12 SIMPLEREGION

This model matches the memory region the operation is assigned to. This can be used to account for different energy consumptions, depending on whether an instruction is executed from, e.g., Flash or SPM.

Listing 11.13: SIMPLEREGION EnergyModel

```
<case><ID>SIMPLEREGION</ID>
  <region>PFLASH-NC</region> <!-- The region name, as defined in the memory layout -->
  <energy>10.0</energy> <!-- Energy to add -->
  <cumulative>true|false</cumulative> <!-- optional parameter, defaults to true -->
</case>
```

11.3.13 SUBTREE

This is a special model, which can basically hold another energy model. All cases within the SUBTREE block are matched, as if they were their own energy specification (i.e., deepest first match). Then the energy consumption of this meta-block is added to the operation under test.

Listing 11.14: COREID EnergyModel

```
<case><ID>SUBTREE</ID>
  <cumulative>true|false</cumulative> <!-- optional parameter, defaults to true -->

  <case><ID> ...</ID>
    ...
  </case>

</case>
```

11.4 ACEC Estimators

Estimating the execution counts of each basic block is completely decoupled from the amount of energy which a basic block's or an instruction's execution will need. Typically, the *average*-case execution counts (ACEC) are probably calculated, but the framework could easily be extended to provide worst-case execution counts if needed.

Warning

At the time of writing this documentation, all estimators are still quite new and untested. Do not blindly rely on their results. If using this framework for a publication or thesis, always verify the provided results!

11.4.1 ACEC Estimation by Heuristic

The heuristical ACEC estimator class ACECHeuristic tries to give a statistical measure on the average-case exeuction count of each basic block.

For this, it is assumed that no knowledge about the programs input parameters and the program's logic is known. As a result, it is assumed that at each conditional jump (e.g., an **if** statement), the probability of executing the explicit jump target is 0.5. The initial basic block of an entrypoint is executed with a probability of 1. Following this scheme, the probability of each basic block's execution is calculated.

For loops, the execution probability of the basic blocks within the loop are multiplied by the average-case execution frequency of the loop. This is taken from the user's loopbound annotations which are needed for WCET analysis. If the minimum and maximum loopbound are equal, these are taken as average-case loop bound. If the minimum and maximum loopbound differ, the arithmetic mean is taken.

Warning

The current implementation of the heuristic may lead to cases where the probabilities are not exact. This may happen, if multiple branches of a program re-unite at a later point.
Please be aware, that this estimator currently only provides an approximation of the execution probability.

The ACEC estimation using the heuristic is done for each LLIR_Entrypoint individually. Therefore, multi-entrypoint programs can be analyzed using this estimator without any limitations.

11.4.2 ACEC Estimation by Simulation

WCC is tightly coupled to the Synopsys CoMeT platform simulator. This simulator can thus be used in order to estimate the average-case execution count of each basic blocks of a program.

At the current state of the implementation, when using this estimator the program under analysis is compiled and executed by CoMeT starting at its main() function. The program is then executed once and the execution counts of each basic block during this execution are counted.

Be advised, that this implies that the program features user-provided exemplary input data which lead to a typical program execution. If a basic block is never executed in this single simulation run, its ACEC will be 0 and any subsequent energy analysis/optimization may assume that the block is *never* executed.

Also, WCC's current CoMET integration does not support multiple entrypoints. If entrypoints do not feature shared code, a dummy main() may be created which calls each entrypoint once. However, *if* entrypoints use shared code, there is no way of providing safe average-case execution count estimations in the context of each entrypoint.

11.4.3 ACEC Estimation by WCET Analysis

This ACEC estimator uses aiT in order to estimate the execution counts of the basic blocks. This can be used to provide some worst-case-like energy estimate.

The idea behind is the assumption, that the worst-case execution path equals the path which also features the highest energy consumption.

Due to the functioning of WCET analysis' IPET based path analysis, the definition of the worst-case execution count differs from the definition of the average-case execution count of a basic block: The worst-case execution count denotes the number of times the basic block is executed *if* it is part of the worst-case execution path. This means, that a basic block may easily have a worst-case exeuction count of zero, despite the fact that it can be executed.

Also, the worst-case execution path through a program may very likely not be a "typical" path through the program. It is up to the user to decide if, why and how useful this estimator is.

11.5 Performing the Energy Estimation

This section gives an overview of the technical implementation of the energy analyzer. It is not needed in order to *use* the energy analysis, and it is not strictly needed in order to *extend* the energy analysis model. Yet, it should provide some insight for people trying to work at the code of the analyzer.

From a class-structure based point of view, the framework consits of platform independent and platform dependent parts. The platform-*independent* classes are based under LLIROPTIMIZATIONS/lliranalyses/energy. The platform-*dependent* classes are based under LLIROPTIMIZATIONS/llir*ARCH*analysis/energy. At the date of writing this text, the framework was mainly used for the ARM target. Thus, descriptions of the platform-dependent aspects will also implicitly focus on ARM.

Note

This documentation should give some introduction into the class design of the energy analysis framework. However, it is very likely that it will not be maintained perfectly over time. Thus, do not rely on this information to be complete and/or up-to-date. If in doubt, better read each class' doxygen documentation and/or code in addition or instead.

11.5.1 General Structure

The analysis framework consists of two fundamental steps: Firstly, the XML specification is read and an analysis-tree is created. Each node in this tree represents an EnergyModel from the XML file (cf. Section 11.3). The structure of the tree represents the structure of the XML model with left-to-write matching top-to-bottom in the XML. I.e., anything the first EnergyModel in the XML file will be at the outmost left in the analysis tree. A nested EnergyModel in the XML will result in a child node in the analysis tree The graph ist directed and the direction of the interconnects between the nodes represents the nesting level. I.e., the direction of the interconnects points from outer EnergyModels to the nested ones.

The second fundamental step of the framework is the analysis itself. This, again, is split in three parts: The Estimation of the energy consumption needed for executing one given LLIR_Instruction with one given predecessor once. This step does not yet honor whether the given instruction *is* actually executed. It only determines, that *if* the instruction is executed, then it would need x energy.

The second part of the analysis is the execution count estimation. There, the number of times each basic block is executed in the context of a given entrypoint is calculated. Refer to Section 11.4 for the currently supported different estimation techniques.

The last part of the analysis is the combination of execution counts and energy consumption. The results are attached to the LLIR_BBs, LLIR_Functions and LLIRs as objectives.

In the end, each LLIR_BB has an objective which accounts for the energy consumption needed to execute this basic block *once*. The LLIR_Function has an objective which accounts for the energy consumption needed to execute this function *once*. I.e., here, the energy consumption of each basic block are multiplied with their respective average-case execution count per function call. Nested function calls are *not* accounted for. The LLIR contains an objective which accounts for the energy consumption needed to execute all functions within this LLIR in an average-case scenario. I.e., the energy consumption of

each function is multiplied by the function's average-case call frequency.

11.5.2 XML Parser and Graph Structure

The XML file is pares using LIBBOOST property tree classes. Parsing an XML file is platform independent. However, the supported EnergyModel classes are very likely at least partially platform dependent. Therefore, the XML parser is placed under the platform dependent analysis directory. For ARM7, it is ARM7EnergyGraph.

The class parses the supplied XML file from top to bottom. This is done recursively in order to handle an arbitrary number of nested blocks. For each XML block specified by the user, the appropriate EnergyModel class is created. Memory management is done by using C++ shared pointers.

Then, the newly created object is inserted as a node into the analysis graph. The graph class itself i called EnergyGraph and is platform-independent. Each node is an EnergyModel. The graph may have multiple top nodes. For each node, the children may be retrieved. Additionally, the graph features a public member function writedot(...) which writes the graph as a dot file to a std::ofstream.

11.5.3 EnergyModels

Each EnergyModel is based on the abstract base class ModelInterface. This provides the interface of each energy model.

Due to this common interface, the analysis graph can be completely unaware of the actual model which is analyzed. EnergyModels may be added or adapted as needed, without any changes to the analysis itself. Refer to the doxygen documentation of the ModelInterface class for further information.

A fully initialized EnergyModel may receive a TaskEntry, a LLIR_Operation and a preceeding LLIR_Operation. It then returns the (partial) energy consumption of the operation with the provided predecessor. Due to first-deepest-match idea of the analysis tree, multiple models may match, and their returned energy costs may be either added or substituted. Re-read this chapter, if you are confused by this. If the EnergyModel does not match the provided operation for whatever reason, 0 is returned as energy consumption.

11.6 Extending the Analyzer

This section gives a short introduction on the steps which might be needed in order to extend the analyzer. Note, that this is *not* a full step-by-step tutorial. Addition of new files and classes, as well as necessary adjustments to the build system and similar stuff will not be covered.

11.6.1 Extending the Energy Specification

Extending the energy specification is usually identical to creating a new EnergyModel and adding it into the analysis framework.

1. Think about the XML syntax. Think of a new `ID`, the properties for the XML block,... Write a dummy XML file or extend your current XML file by the new block.

2. Create your new EnergyModel class. Think about whether the class is platform-dependent or independent. Depending on your decision, put your files into lliranalyses/energy or llir*ARCH*analyses/energy. The class *must* be publicly inherit from the ModelInterface base class. All XML properties from the energy specification file should be passed to your class' constructor. Do not introduce any public init() function or similar. Your new class' objects must be ready to use right after creating the object. Refer to ModelInterface doxygen documentation for more information on how to implement the member functions defined by the interface.

3. Now all that's left is to integrate the new class into the XML parser. Open (for ARM7) llirARManalyses/energy/arm7energygraph.cc Find the function generateNode(...). You will notice a large **if**(...)... **else if**(...)... structure. Add a new **else if** condition which matches your `ID`. Read the mandatory or optional XML properties as needed and create an instance of your class using shared_ptr. Compare with the existing cases like `OPCODE` for reference. Note, that the conditionals are alphabetically orderd in the source code.

4. Compile WCC and try to parse your XML file. Turn Debugging output on as needed. The new EnergyModel should now fully work.

5. Add your new EnergyModel to this documentation.

11.6.2 Adding a New ACEC Estimator

This section briefly describes which steps are needed in order to integrate a new execution count estimator.

1. Think about whether the execution-count estimator is platform dependent or independent. Depending on your decision, put your files to lliranalysis/energy or llir*ARCH*analysis/energy. In the respective directory, create a new class which publicly inherits form ACECInterface.

2. Refer to the doxygen documentation of ACECInterface and compare with an existing execution count estimator to implement the public member functions of the interface class.

3. The execution count estimator can be chosen by the user on WCC's command line. Edit wccoptions.cc and configuration.cc accordinlgy in order to set and get appropriate values for the Configuration::ENERGY_ACEC_ESTIM_MODE configuration property.

4. The execution count estimator is created in the analysis' constructor. Therefore, go to (for ARM7) llirARManalysis/energy/arm7energyanalysis.cc. Find the constructor and extend the **if**(...)... **else if**(...)... block which selects the execution count estimator (member variable mAcecEstimator) appropriately.

5. Compile WCC, check that the command line help shows your new entry, and test whether setting the ACEC estimator to your newly introduced value correctly activates your new execution count estimator.

6. Document your new execution count estimator in this documentation.

12 Synopsys CoMET Instruction Set Simulator

CoMET is a commercial software of Synopsys that allows instruction-set simulation of code for configurable hardware platforms. Within WCC, CoMET is used for various purposes, including

- Profiling, i.e., obtaining "average-case" figures on execution counts, execution times, memory accesses, cache events etc.
- ACET analysis.
- Multi-Core platform design, i.e., the ARM7-based multi-core architecture described in Chapter 8 is realized using CoMET.
- Testing, i.e., WCC's TESTBENCH can be configured to use CoMET for the validation of all test cases.

This chapter covers some technical aspects of how to pre-compile WCC's CoMET modules and platforms.

12.1 Overview

CoMET's nomenclature distinguishes between modules and platforms. A *module* is a single hardware component that could be instantiated in order to build up complete architectures, e.g., a processor core with a certain ISA, or a memory, or a cache, or a bus. Modules can then be taken and instantiated (several times if desired) so that larger, complete systems can be created. Such complete systems are then finally called *platforms*. CoMET makes use of SystemC in order to specify modules and platforms.

In order to keep the simulation speed high, CoMET exploits the principle of compiled simulation, i.e., all the modules and platforms are pre-compiled into binary shared libraries on your *simulation host system*. CoMET comes with a variety of existing modules, some of them being pre-compiled, closed source IP cores due to license restrictions (e.g., the processor modules for ARM or Infineon CPUs), some of them are open.

All the modules and platforms that WCC uses internally reside in the directory `trunk/WCC/COMET`. For the aforementioned reasons, this directory contains both the SystemC source codes and the pre-compiled binary libraries of the modules and platforms, as well as individual project and configuration files. The following section describes how to re-build all the pre-compiled libraries for WCC's CoMET modules and platforms if, e.g., a migration to a new CoMET version is required.

12.2 Building WCC's CoMET Modules and Platforms

Note

The following steps have been performed for and validated with CoMET version K-2015.12. For other CoMET versions, individual steps described in the following might have to be adapted.

1. Change your file `$HOME/.module.linux` to include your CoMET version, e.g., add `+comet/K-2015.12` and remove any other CoMET version. Log out and in again.
2. *Optional: Should you run into license errors displayed by CoMET in some of the subsequent steps, rename your file* `$HOME/.bashrc_user` *so that it does not get included. Log out and in again.*

The following sections individually describe the required steps for WCC's ARM and TriCore CoMET modules and platforms.

12.2.1 ARM

Modules

WCC's hardware modules that are used to build ARM-based CoMET platforms reside in directory `trunk/WCC/COMET/lib`. In order to build these modules from scratch, proceed as follows:

3. Start the CoMET GUI: `comet.sh &`
   During startup, simply hit "OK" if CoMET asks for workspace directories or subversion plugins.
4. Include pre-installed commercial processor and peripheral modules:
   a) Window / Preferences / VaST / Modules Database / User Module Locations (Installation):
   Choose
   `/opt/local/synopsys/pa/K-2015.12/Fast_Timed_IP/modules/arm/processors/arm7tdmi/v4.12.0`
   as root directory and confirm.
   b) Repeat the previous step also for directories
   `/opt/local/synopsys/pa/K-2015.12/Fast_Timed_IP/modules/arm/peripherals/arm_vic_pl190`
   `/opt/local/synopsys/pa/K-2015.12/Fast_Timed_IP/modules/interconnect/stdbus_bridge`
   c) VaST / Rescan Module Database
5. Build WCC's CoMET modules:

a) File / Import / General / Existing Projects into Workspace:
Choose
`trunk/WCC/COMET/lib`
as root directory. Select all projects showing up in the dialog. Do not tick "Copy projects into workspace". Confirm the import dialog.

b) For each project now showing up in the workspace GUI, do:
Project / Build Configurations / Set Active / Debug

c) Build all Projects: Project / Build All

d) WCC's CoMET modules are now all built. Clean up your workspace by removing all these previous projects from the GUI. Do not tick "Delete project contents on disk (cannot be undone)".

Basic ARM7TDMI Platform

6. Make sure that the pre-installed modules from step 4 above are still included.

7. WCC's basic ARM7TDMI platform uses the modules `cache_controller`, `cached_ram` and `const_number_device` just built in the previous step 5. Make sure that they are found by CoMET:

   a) Window / Preferences / VaST / Modules Database / User Module Locations (Installation):
   Choose
   `trunk/WCC/COMET/lib/cache_controller`
   as root directory and confirm.

   b) Repeat the previous step also for directories
   `trunk/WCC/COMET/lib/cached_ram`
   `trunk/WCC/COMET/lib/const_number_device`

   c) VaST / Rescan Module Database

8. Build WCC's `ARM_7TMDI` CoMET platform:

   a) File / Import / General / Existing Projects into Workspace:
   Choose
   `trunk/WCC/COMET/PLATFORM_ARM_7TDMI`
   as root directory. Select all projects showing up in the dialog. Do not tick "Copy projects into workspace". Confirm the import dialog.

   b) For each project now showing up in the workspace GUI, do:
   Project / Build Configurations / Set Active / Debug

   c) Build all Projects: Project / Build All

9. After this step, the complete CoMET hardware platform resides in directory `trunk/WCC/COMET/PLATFORM_ARM_7TMDI/base_arm7tdmi_x1_system`. In this directory, there is a template file `vast.opt.tmpl.in` that might require some manual

post-processing. Open this file in a text editor and carefully check all the `LoadDll` directives therein. As mentioned in Section 12.1, all the pre-compiled modules and platforms are stored as binary shared libraries. Thus, CoMET has to be able to locate these shared libraries upon simulation. You need to check two different scenarios in `vast.opt.tmpl.in`:

a) If you migrate to a new CoMET version, the paths to the pre-installed modules and platforms might have changed, esp. if version numbers are part of these paths. Thus, check all `LoadDll` directives beginning with `@COMET_INSTALL_PATH` and adjust these paths accordingly.

b) Cross-check that all the required models within `trunk/WCC/COMET` that the CoMET platform depends on are also captured by `LoadDll` directives. In particular, all modules from folder `trunk/wcc/COMET/lib` that you specified in step 7 above must be captured by appropriate `LoadDll` directives beginning with `@abs_srcdir@`.

   Furthermore, the platform itself features several shared libraries that reside in the sub-directories `base_arm7tdmi_x1`, `base_arm7tdmi_x1_system` and `metrix_trace` under path `trunk/WCC/COMET/PLATFORM_ARM_7TDMI`. Make sure that all these shared libraries are also properly covered within file `vast.opt.tmpl.in`.

c) Since `vast.opt.tmpl.in` is just a template from which the actual `vast.opt.tmpl` configuration file that the CoMET simulator will definitely use is generated, you might have to re-run WCC's global `configure` script now (cf. Section 2.1), or you need to port all the changes that you applied to template `vast.opt.tmpl.in` manually to `vast.opt.tmpl`.

d) Should CoMET ever complain during simulation that it is unable to find some required shared libraries, then it is very likely that some `LoadDll` directives in `vast.opt.tmpl.in` refer to incorrect directories or are simply missing.

ARM-Based Multi-Core Platforms

The following steps apply to all ARM-based Multi-Core Platforms of WCC, i.e., to all directories `trunk/WCC/COMET/PLATFORM_ARM_MC_X*`. For the sake of simplicity, the remainder of this section only refers to `PLATFORM_ARM_MC_X8`.

10. Make sure that the pre-installed modules from step 4 above are still included.

    For `PLATFORM_ARM_MC_X1`, the root directory
    `/opt/local/synopsys/pa/K-2015.12/Fast_Timed_IP/modules/arm/peripherals/arm_ipcm_pl320`
    also has to be added.

11. WCC's ARM 8-core platform uses almost all modules from `trunk/WCC/COMET/lib` that have been built in step 5. Make sure that they are found by CoMET:

a) Window / Preferences / VaST / Modules Database / User Module Locations (Installation):
Choose
`trunk/WCC/COMET/lib/arbitration_bridge`
as root directory and confirm.

b) Repeat the previous step also for directories
`trunk/WCC/COMET/lib/base_arm7tdmi_x1`
`trunk/WCC/COMET/lib/cached_ram`
`trunk/WCC/COMET/lib/cache_controller`
`trunk/WCC/COMET/lib/const_number_device`
`trunk/WCC/COMET/lib/global_memory`

c) VaST / Rescan Module Database

12. Build WCC's `ARM_MC_X8` CoMET platform (cf. step 8).

13. Manually edit the platform's `vast.opt.tmpl.in` template configuration file (cf. step 9).

12.2.2 Infineon TriCore

14. Include pre-installed commercial processor and peripheral modules:

a) Window / Preferences / VaST / Modules Database / User Module Locations (Installation):
Choose
`/opt/local/synopsys/pa/K-2015.12/Fast_Timed_IP/modules/infineon/processors/tc1_3_1v1`
as root directory and confirm.

b) Repeat the previous step also for directories
`/opt/local/synopsys/pa/K-2015.12/Fast_Timed_IP/modules/infineon/processors/pcp2`
`/opt/local/synopsys/pa/K-2015.12/Fast_Timed_IP/modules/infineon/peripherals/tricore_sbcu`
`/opt/local/synopsys/pa/K-2015.12/Fast_Timed_IP/modules/infineon/peripherals/tricore_lbcu`
`/opt/local/synopsys/pa/K-2015.12/Fast_Timed_IP/modules/infineon/peripherals/tricore_lfi_bridge`
`/opt/local/synopsys/pa/K-2015.12/Fast_Timed_IP/modules/infineon/peripherals/tricore_interrupt_bus_model`

c) VaST / Rescan Module Database

15. Build WCC's `PLATFORM_TRICORE_TC1797` CoMET platform:

a) File / Import / General / Existing Projects into Workspace:
Choose

`trunk/WCC/COMET/PLATFORM_TRICORE_TC1797`
as root directory. Select all projects showing up in the dialog. Do not tick "Copy projects into workspace". Confirm the import dialog.

b) For each project now showing up in the workspace GUI, do:
Project / Build Configurations / Set Active / Debug

c) Module `scd_vast_systemc_simple_srn` depends on SystemC versions 2.1 or 2.2 that come pre-installed with CoMET. Before building the module, make sure that this SystemC distribution is actually used by adapting the project's build flags:

i. Right-click the project `scd_vast_systemc_simple_srn` in the CoMET GUI, select "Properties" from the context menu.

ii. Choose C/C++ Build / Environment

iii. Double-click entry `SYSTEMC` and set its value to `/opt/local/synopsys/pa/K-2015.12/SLS/virtualizer-comet/linux/systemc_v2.2`

iv. Confirm with OK / Apply / OK

d) Build all Projects: Project / Build All

16. Manually edit the platform's `vast.opt.tmpl.in` template configuration file (cf. step 9).

13 Miscellaneous

This chapter will contain a collection of miscellaneous stuff which may be interesting for WCC users or developers. This comprises usage information for analyses, optimizations, ...