Bitwise Identical Compiling Setup: Prospective for Reproducibility and Reliability of Earth System Modeling

R. Li[2,1], L. Liu[1,3], G. Yang[2,1,3], C. Zhang[2,1] and B. Wang[1,3,4]

[1]{Ministry of Education Key Laboratory for Earth System Modeling, Center for Earth System Science (CESS), Tsinghua University, Beijing, China}

[2]{Department of Computer Science and Technology, Tsinghua University, Beijing, China}

[3]{Joint Center for Global Change Studies (JCGCS), Beijing, China}

[4]{State Key Laboratory of Numerical Modeling for Atmospheric Sciences and Geophysical Fluid Dynamics (LASG), Institute of Atmospheric Physics, Chinese Academy of Sciences, Beijing, China}

Correspondence to: L. Liu (liuli-cess@tsinghua.edu.cn), G. Yang (ygw@tsinghua.edu.cn)

Abstract

Reproducibility and reliability are fundamental principles of scientific research. A compiling setup that includes a specific compiler version and compiler flags is an essential technical support for Earth system modeling. With the fast development of computer software and hardware, a compiling setup has to be updated frequently, which challenges the reproducibility and reliability of Earth system modeling. The existing results of a simulation using an original compiling setup may be irreproducible by a newer compiling setup because trivial round-off errors introduced by the change of compiling setup can potentially trigger significant changes in simulation results. Regarding the reliability, a compiler with millions of lines of code may have bugs that are easily overlooked due to the uncertainties or unknowns in Earth system modeling. To address these challenges, this study shows that different compiling setups can achieve exactly the same (bitwise identical) results in Earth system modeling, and a set of bitwise identical compiling setups of a model can be used across different compiler versions and different compiler flags. As a result, the original results can be more easily reproduced; for example, the original results with an older compiler version can be reproduced exactly with a newer compiler version. Moreover, this study shows that new test cases can be generated based on the differences of bitwise identical compiling setups between different models, which can

help detect software bugs in the codes of models and compilers and finally improve the reliability of Earth system modeling.

1 Introduction

Earth system modeling simulates interactions between components of the climate system (e.g., atmosphere, oceans, land surface, sea ice, etc.). It plays a critical role in understanding the past and present climate, and in predicting future climate. An increasing number of models have sprung up all over the world, including stand-alone component models and coupled models consisting of multiple component models, such as Climate System Models (CSMs) and Earth System Models (ESMs).

The development of models for Earth system modeling heavily depends on the advancement of computer supports, not only in terms of hardware (such as high-performance computers) but also in terms of software such as compiling setups that include compiler versions and compiler flags. During the continuous evolution of the models, the compiling setups have to be updated frequently for the compatibility of newer high-performance computers with new processors and for better computing performance.

One may think it is easy to update compiling setups, just by installing new compiler version or changing compiler flags. However, it is challenging to update compiling setups for Earth system modeling, because researchers may get significantly different results from the same experiment when using different compiling setups (Liu et al., 2015b). A compiler not only translates the code in a high-level programming language to a low-level language but also tries to improve computational performance of the codes with compiler optimization schemes. Compilers from different families (for example, those in Table 1) and different versions from the same compiler family generally differ in performance optimization schemes as well as the corresponding implementations. On the other hand, different compiler flags of the same compiler version enable and disable different sets of performance optimization schemes. That is why different compiling setups can lead to different results of the same program. The updating of compiling setups therefore will introduce at least two challenges to Earth system modeling. The first challenge concerns reproducibility of simulation results. Due to the chaotic nature of the climate system, more and more studies have shown that trivial round-off errors can trigger significant changes in simulation results of Earth system modeling (Hong et al., 2013; Liu et al., 2015b; Song et al., 2012). Due to the differences of performance optimization schemes among different compiling setups, a change of compiling setups potentially introduces round-off errors. As a

result, the results of a simulation obtained with a compiling setup may be irreproducible by another compiling setup.

The second challenge is the reliability of the simulation results. Compilers are large-scale programs with millions of lines of code. It is well understood that with more lines of code there are more potential bugs in the program. Therefore, although there is generally a large amount of software testing before releasing a compiler version, there still can be unknown bugs. Models for Earth system modeling are also large-scale numerical programs with more and more lines of code (Easterbrook and Johns, 2009). There are already ESMs with nearly one million lines of codes (Alexander and Easterbrook, 2015). Therefore, it is possible that some bugs in a compiler version may be triggered by some code segments in a model.

In response to these challenges, several issues about compiling setups should be investigated:

1) Can different compiling setups achieve the same (bitwise identical) simulation results? If yes, it will be much easier to reproduce previous simulation results.

2) How can compiler flags to be selected to compile the codes of a model. Since a compiler version always contains many performance optimization schemes, there are a lot of choices of compiler flags.

3) How to determine whether compiler bugs are triggered in a model simulation. If compiler bugs can be detected, researchers can modify the code to avoid the compiler bugs or select a "safer" compiling setup. Compiler bugs are very difficult to detect, especially when they do not lead to a crash of the simulation. There are many uncertainties and unknowns in Earth system modeling, so compiler bugs can easily be overlooked due to these uncertainties or unknowns.

There are already efforts for the above-mentioned issues. It has been demonstrated that with a certain compiler flag, different compiler versions can achieve bitwise identical simulation results for a given model (Liu et al., 2015a). But it is still not known whether compiling setups with the same compiler version but different compiler flags can achieve bitwise identical simulation results. In this paper, we call the compiling setups that can achieve bitwise identical simulation results "bitwise identical compiling setups." It is also unknown whether the bitwise identical compiling setups of one model are appropriate for another model. Baker et al. (2015) proposed a new ensemble-based consistency test for the Community Earth System Model (CESM; Hurrell et al. 2013). It can effectively verify whether two compiling setups can achieve

consistent simulation results, especially when they do not achieve bitwise identical simulation results. However, we cannot be sure whether a compiling setup is right or wrong. In other words, it cannot help detect compiler bugs. As a result, it is possible that a compiling setup with compiler bugs has been used for the development of a model for a number of years, while a new compiling setup with bug fixes cannot be used for the model development due to the failure in consistency tests.

The results in this paper show that the bitwise identical compiling setup sets of a model can extend to different compiler versions and different compiler flags. They can facilitate the reproduction of original simulation results, assist researchers to determine the compiler flags for model simulations, help researchers build more test cases to detect bugs in models and compilers, and finally improve the reproducibility and reliability of Earth system modeling.

The rest of this paper is organized as follows. Section 2 briefly introduces compiler optimizations. Section 3 shows the bitwise identical compiling setups of three models. Section 4 uses examples to show what can be learned from the comparison of bitwise identical compiling setups between different models. We conclude this paper with discussion in Section 5.

2 Brief introduction to compiler optimizations

Models for Earth system modeling are generally programmed in languages such as Fortran, C and C++. A number of compilers have been used for Earth system modeling, such as the compiler families listed in Table 1. In the following context, we further introduce the Intel compiler family and GNU Compiler Collection (GCC) with details.

The Intel compiler family, which is developed by the Intel Corporation, is a commercial software product. It has been widely used for Earth system modeling, because most of the high-performance computers for Earth system modeling are equipped with the CPUs manufactured by the Intel Corporation. Table 2 shows the five latest Intel compiler versions (from version 11.1 released in 2009 to version 15.0.1 released in 2014). For each compiler version, there are many compiler optimization options. Table 3 shows several compiler optimization options that may impact the precision of floating-point calculation. They are common to all compiler versions listed in Table 2. For a compiler flag such as “-fp-model”, there may be multiple selections of the values.

GCC is the most widely used free compiler family in the world. Table 4 shows the five latest GCC versions (from version 4.6.4 released in 2013 to version 5.1 released in 2015). For each compiler version, there are also many compiler optimization options. Similar to Table 3, the compiler optimization options in Table 5 may impact the precision of floating-point calculation and are common to all GCC versions listed in Table 4.

3 Bitwise identical compiling setups

In this study, we use three models, namely, CAM5 (Neale et al., 2010), POP2 (Smith et al., 2010) and FGOALS-g2 (Li et al., 2013a). To obtain bitwise identical compiling setups of a given model, we should first design various compiling setups and then run the model using each of them. In this section, we will briefly introduce the three models, the compiling setups and the bitwise identical compiling setups of each model.

3.1 Models and simulations

The version of CAM5 used in this study is CAM5.3. It is released as the atmosphere component of the CESM version 1.2 (CESM1.2). It contains more than 550,000 lines of source code mainly programmed in Fortran. It can be used as a standalone model or the atmospheric component of CESM1.2. In this study, we use CAM-5.3 as a standalone model. CAM-5.3 supports different dynamic cores and different resolutions. To run the standalone CAM-5.3, we use the default setting (details can be found at *http://www.cesm.ucar.edu/models/cesm1.2/cam/docs/ug5_3/ug.html*), where the dynamic core is finite volume and the resolution of the horizontal grid is 1.9°x2.5°.

POP2 used in this study is the ocean component of CESM-1.2. It is based on the POP version 2.1 of the Los Alamos National Laboratory. It contains more than 170,000 lines of source code mainly programmed in Fortran. To run POP2 as a standalone model, we use the component set "*C_NORMAL_YEAR*" of CESM-1.2, which uses POP2 as the ocean component and the other components as data models. The horizontal grid selected is marked as "*T62_gx1v6,"* while the other settings of the simulation are default.

FGOALS-g2 is a fully coupled CSM consisting of the atmosphere model GAMIL2 (Li et al., 2013b), ocean model LICOM2 (Liu et al., 2004), land surface model CLM3 (Oleson et al., 2004), and an improved version (Wang et al., 2009; Liu, 2010) of the sea ice model CICE4 (http://climate.lanl.gov/Models/CICE). It participated in the the Coupled Model Intercomparison Project Phase 5 (CMIP5) and is widely used for scientific research. It contains

about 240,000 lines of source code mainly programmed in Fortran. GAMIL2 and CLM3 use the same horizontal grid whose resolution is about 2.8 °, while LICOM2 and CICE4 uses the same horizontal grid whose resolution is about 1 °. To run FGOALS-g2, we use the CMIP5 pre-industry control (pi-Control) experiment setup.

All simulations of the models are run on the same high-performance computer named Tansuo100 at Tsinghua University in China, which consists of more than 700 computing nodes, each of which consists of two Intel Xeon 5670 6-core CPUs sharing 32GB main memory. Specifically, we use 16, 16 and 17 processes to run CAM5.3, POP2 and FGOALS-g2, respectively.

3.2 Compiling setups

By combining different settings of different compiler optimization options listed in Table 3, there are more than 4,000 possible compiler flags. Considering there are four major optimization levels (O0-O3) in an Intel compiler version, there are more than 16,000 possible compiler flags for an Intel compiler version. Similarly, there are more than 1,000 possible compiler flags for a GCC compiler version.

It is impractical for us to investigate all compiling setups. We decided to use five Intel compiler versions (versions 11.1, 12.1, 13.0, 14.0.1, and 15.0.1) and five GCC compilers versions (versions 4.6.4, 4.7.4, 4.8.5, 4.9.3, and 5.1) for this study, and take into consideration four optimization levels (O0-O3). For a compiler version at an optimization level, we selected a small number of compiler flags (Table 6 for the Intel compilers and Table 7 for the GCC compilers).

3.3 Bitwise identical compiling setups of models

To obtain the bitwise identical compiling setups of a model (CAM5, POP2, or FGOALS-g2), we use each compiling setup (in Section 3.2) to compile the model code and then run the corresponding model simulation. A short integration is enough to check bitwise identity of simulation results (Easterbrook and Johns, 2009). In detail, we use five model days for each simulation and use the binary formatted data file of daily output of fields for bitwise identical comparison. Tables 8-10 show the bitwise identical compiling setups of a model when using the Intel compiler versions, while Tables 11-13 correspond to the GNU compiler versions. In each table, the compiling setups corresponding to the same color (except for white) of

simulation results constitute a bitwise identical compiling setup set of the same model. There is no bitwise identical compiling setup set across the two compiler families.

Given the Intel compiler versions, we can see that there is no bitwise identical compiling setup set between the version 11 and any other version. This is because the version 11 and the subsequent versions use different default instructions to generate the binary code (https://software.intel.com/en-us/forums/intel-visual-fortran-compiler-for-windows/topic/281713), which produces different bitwise results (https://software.intel.com/en-us/forums/intel-visual-fortran-compiler-for-windows/topic/279705).

4 Comparison of bitwise identical compiling setup sets between models

From Tables 8-10 (or Tables 11-13), we can find that, given the same compiler family, bitwise identical compiling setup sets of different models are obviously different. What causes such differences and what can we learn from the differences? To answer these questions, we take the compiling setups of Intel compilers as an example. Based on the results in Tables 8-10, we can generate ideal bitwise identical compiling setup sets (Table 14), following the criterion that if any model achieves bitwise identical results with two different compiling setups, these compiling setups belong to the same ideal bitwise identical compiling setup set. Through comparing Tables 8-10 to Table 14, we can pose a number of questions; for example:

1) Regarding all Intel compiler versions, given compiler flag 2 (or 4), why does CAM5 obtain different simulation results when changing compiler optimization level from O0 (or O1) to O2 (or O3)?

2) Regarding Intel compiler version 13, why does POP2 obtain different simulation results when changing the compiler optimization level from O3 to another level?

3) Regarding Intel compiler version 12, given optimization level O2, why does POP2 obtain different simulation results when changing the compiler flag from 2 (or 3) to 1.

4) Regarding Intel compiler version 13, given optimization level O3, why does POP2 obtain different simulation results when changing the compiler flag from 8 (or 9) to 1?

5) Regarding Intel compiler versions 13, 14 and 15, why does POP2 obtain the bitwise identical results when changing the compiler flag from 1 to 2 (or 4), but CAM5 and FGOALS-g2 do not?

Next, we search for answers to the first two questions, namely, what causes such differences and what can we learn from the differences.

4.1 Methodology

If a code segment can trigger different compiler optimizations under different compiling setups, it may lead to different results in different compiling setups. In the rest of this paper, we call this kind of a code segment "a compilation-sensitive code segment" and call a code file with compilation-sensitive code segments "a compilation-sensitive code file." A model for Earth system modeling generally contains a large number of code segments. To reveal why a model does not achieve bitwise identical results in two different compiling setups, a straightforward way is to find out all compilation-sensitive code segments for further analysis. Given two compiling setups (donated as C_A and C_B) that do not achieve bitwise identical results for a simulation, we propose three stages for the detection of compilation-sensitive code segments:

1) Detect the compilation-sensitive code files. A model generally contains a number of source code files. In the compiling process of a model, we can use C_A to compile a portion of source code files while use C_B to compile the remaining source code files if the object files can be linked together. For example, at the first step, we can use C_A to compile all source code files and then run a simulation to generate a reference result. At the second step, we can divide the source code files into two parts, each of which takes about a half, and then use different compiling setups to compile the two parts (use C_A to compile the first part and use C_B to compile the second part, or use C_B to compile the first part and use C_A to compile the second part). If the result from the same simulation is not bitwise identical with the reference result, the part that is compiled with C_B should contain compilation-sensitive code files, and next we will recursively detect compilation-sensitive code files in that part.

2) Detect compilation-sensitive code segments in a compilation-sensitive code file. We propose to log (in binary format) and then bit-to-bit match the values of the input variables and output variables of each code segment in the two compiling setups (C_A and C_B). A code segment with bitwise identical inputs but different outputs is a compilation-sensitive code segment. The size of a compilation-sensitive code segment should be as small as possible, in order to facilitate further analysis. For a source file containing many lines of code, we can either divide it into several new files of smaller size and then repeat the first and second stages for these new files, or into several big code segments at the first step and then

recursively repeat the second stage for the code segments that are compilation-sensitive. The size of a code segment cannot be too small because the function calls for logging the values of variables may result in changes to compiler optimizations so as to change simulation results. In other words, the splitting of a code file or the inserting of the functions for logging values must keep bitwise simulation results.

3) Analyze why a code segment is sensitive. In this stage, we should read the code to check whether there are bugs. Sometimes, it is necessary to compare the differences of assembly codes of the code segment under the two compiling setups.

Researchers may have to conduct the second and third stages manually. However, for the first stage, we designed and implemented a software tool named CoSFiD, which stands for **Co**mpilation-**S**ensitive code **Fi**le **D**etection tool; it can automatically detect compilation-sensitive code files (Section 4.2).

4.2 The CoSFiD

Figure 1 shows the flowchart of CoSFiD. The inputs include the two compiling setups (C_A and C_B), the rules to compile and run the model, and the rules to compare results at bitwise identical level. The outputs are a list of compilation-sensitive code files. CoSFiD first generates a reference result with the compiling setup C_A to compile all code files. Following the idea of the first stage introduced in Section 4.1, CoSFiD compiles and runs the model many times and alternatively changes the compiling setup between C_A and C_B for some code files each time.

The biggest challenge to the design and implementation of CoSFiD is how to control the compilation process of each code file. A straightforward approach is to develop a common tool that can successfully compile any model. However, this approach seems impractical because different models may have different systems to compile the code, for example, using different ways to specify code files and different ways to generate header files. We therefore propose to use the original compiling system of a model and design a compiler wrapper accordingly. The compiler wrapper is some script in CoSFiD, which can replace the original compiler commands used for compiling the model. For example, given that a model uses the Intel compiler commands (i.e., icc, icpc and ifort) to compile the code, users should generate pseudo compiler commands with the same names (i.e., icc, icpc and ifort) under a directory through symbolic linking or copying the compiler wrapper of CoSFiD, and then add the directory to the beginning of the corresponding environment variable (for example, “PATH”) of the operating system to

make the pseudo compiler commands used for the compilation of the code, and then replace the compiler flag for compiler optimizations by a label "-DCoSFiD." When compiling a code file, CoSFiD first gets the name of the file through the compiler wrapper; it then looks up the current compiling setup for the file before switching the compiler version to the specified one if necessary and using the specified compiler flag to replace the label "-DCoSFiD"; it finally compiles the code file.

4.3 Examples

4.3.1 Example 1

In this example, we search for the answer to the first question in Section 4: *regarding all Intel compiler versions, given compiler flags 2 or 4, why does CAM5 obtain different simulation results when changing compiler optimization level from O0 or O1 to O2 or O3*? (as shown in Table 8) Following the methodology in Section 4.1, we first generate the two compiling setups *C1* and *C2* using the Intel compiler version 13, compiler flag 2 (*-fp-model precise -fp-speculation=strict -mp1 -no-vec –no-simd*) and two optimization levels (*O1* and *O2*);next, we use CoSFiD to find only one compilation-sensitive code file (*modal_aero_rename.F90*) from more than 700 code files of CAM5. For further analysis, we split *modal_aero_rename.F90* into two temporary code files, each of which contains only one subroutine, and then use CoSFiD to find that only the first subroutine (*modal_aero_rename_sub*) contains compilation-sensitive code segments. Through logging and then comparing the values of input and output variables of code segments in the two compiling setups, we find a compilation-sensitive code segment, shown in Fig. 2. Given the same input (bitwise identical), this code segment can generate slightly different results in different optimization levels (for example, Table 15). This is due to the differences in assembly codes (Table 16). For the exponent *onethird* in Fig. 2, it is defined as *1.0_r8/3.0_r8* in the program. The compiler optimization level *O1* will call function *pow* to calculate the corresponding power function, while *O2* will intelligently find that the power function is actually a cube root operation and then call *cbrt* for the calculation.

After replacing variable *onethird* with *(1.0_r8/3.0_r8)* throughout the code, CAM5 achieves bitwise identical results with compiler flag 2 or 4 throughout all compiler optimization levels, and finally the corresponding bitwise identical compiler setup sets of CAM5 are enlarged. For example, the bitwise identical compiling setup set in green color and the set in blue color in Table 8 are unified into one set.

4.3.2 Example 2

In this example, we search for the answer to the second question in Section 4: r*egarding Intel compiler version 13, why does POP2 obtain different simulation results when changing the compiler optimization level from O3 to another level?* (as shown in Table 9) To generate the two compiling setups *C1* and *C2*, we use the Intel compiler version 13, compiler flag 1 (*-fp-model strict -fp-speculation=strict -mp1 -no-vec –no-simd*) and two compiler optimization levels (*O2* and *O3*). Using CoSFiD, we find only one compilation-sensitive code file (*hmix_gm.F90*) from more than 500 code files of POP2. *hmix_gm.F90* contains about 10 subroutines and about 4,000 code lines. For further analysis, we split *hmix_gm.F90* into 10 temporary code files, each of which contains only one subroutine, and then use CoSFiD again to find that only the temporary code file with the second subroutine (*hdifft_gm*) contains compilation-sensitive code segments. Based on the binary values of input and output variables of the code segments with the two compiling setups, we find a compilation-sensitive code segment in the subroutine *hdifft_gm*, shown in Fig. 3. It is curious that given exactly the same inputs, variable *WORK3* obtains significantly different results in the two compiling setups (for example, Table 17). A manual result (Table 17) confirms correctness of the result in the compiling setup with optimization level *O2*, but indicates that the code segment in Fig. 3 triggers a bug in the compiler when the compiler optimization level is *O3*.

It is almost impossible for us to fix a compiler bug. However, we can try to make the model code not trigger the bug. Further analysis with assembly codes shows that the compiler performs an optimization of loop fusion that merges four two-level loops at lines 1920-1999 of the code file *hmix_gm.F90* into one loop. We intuitively guess that there are bugs in the loop fusion optimization. To avoid the loop fusion optimization, we move the four two-level loops into a new subroutine. Finally, POP2 achieves bitwise identical results with compiler flag 1 throughout all compiler optimization levels, and the corresponding bitwise identical compiling setup sets of POP2 are enlarged. For example, the bitwise identical compiling setup set in red and the set in green in Table 9 are unified into one set.

5 Discussion and conclusion

This study illustrates that a model can achieve bitwise identical results under different compiling setups. For a given model, there are always a number of bitwise identical compiling setup sets, some of which can be across not only different compiler flags but also different versions of the same compiler family. As a result, the original results with an older compiler

version can be exactly reproduced with a newer compiler version. Moreover, the examples in this paper reveal that bitwise identical compiling setup sets can be enlarged through carefully modifying compilation-sensitive code segments, which will facilitate the exact reproduction of original simulation results.

During the development of a model, the model codes increase continuously and need to be tested frequently. The testing can be classified into two categories: scientific testing and technical testing. Scientific testing, which is evaluating the scientific meaning of simulation results, is generally expensive, because it always requires long simulations and requires scientists to evaluate a large amount of results. In contrast, technical testing, which does not depend on the scientific meaning of simulation results, is generally cheap. For example, short simulations (such as several model days) are enough for bitwise identical testing, and bitwise identical testing can be conducted automatically without any burden to scientists (Easterbrook and Johns, 2009). Technical testing therefore should be much more frequent than scientific testing. Since a bitwise identical compiling setup set contains a number of compiling setups that should achieve exactly the same results for a model simulation, it can bring more cases for technical testing. For example, given that a new code version evolves from an old code version with new modifications, the bitwise identical compiling setup sets of each code version can be obtained automatically. If the two code versions do not have the same bitwise identical compiling setup sets, new test cases can be generated for checking why this happens, for example because of bugs in the codes or compilation-sensitive code segments. If there are compilation-sensitive code segments in the new modifications, we advise researchers to make them insensitive, to make each bitwise identical compiling setup set as big as possible for further development of the model. The first example in Section 4.3 reveals that a compilation-sensitive code segment can become insensitive after a slight code modification.

Although the bitwise identical compiling setup sets of different models are generally different, the differences can effectively bring more test cases to detect software bugs in model simulations, especially the bugs of compilers. Although scientists of Earth system modeling generally cannot modify the code of a compiler to fix a bug, they can modify the code of a model to make sure that the model code will not trigger a compiler bug again. For example, based on the differences of bitwise identical compiling setup sets among different models (CAM5, POP2 and FGOALS-g2), we found that a code segment of POP2 triggers a bug of the

Intel compiler version 13, and the compiler bug will not be triggered again with a slight modification to the code segment.

There are generally a large number of choices of compiler flags. Researchers may tend to select a compiler flag that can achieve the best computation performance for a model simulation. Our performance evaluation shows that the compiler flag 3 can achieve the best computation performance among the compiler flags in Table 6. According to Tables 8-10, the bitwise identical compiling setup set corresponding to compiler flag 3 is small. It is already known that climate simulation results can be sensitive to round-off errors. To make simulation results most easily reproduced, researchers may be able to use the compiler flag of the best computation performance in a bigger bitwise identical compiling setup set for a model simulation, when the change of compiler flags will not significantly decrease the computation performance. For example, researchers can use the compiler flag "*-O3 -fp-model strict -fp-speculation=strict -mp1 -no-vec –no-simd*" for the simulation of the atmosphere models CAM5 and GAMIL2 when the Intel compilers are used, because such a compiler flag does not significantly decrease the computation performance, especially when the number of processes is big (Fig. 4 and Fig. 5). Please note that any selection of a compiler flag for a model simulation will not affect the code testing based on bitwise identical compiling setup sets.

Code availability

1. The source code of CESM version 1.2 can obtained at http://www.cesm.ucar.edu/models/cesm1.2/.
2. The source code of FGOALS-g2 is currently not publicly available. You can contract us for more information.
3. The source code of CoSFiD is available at https://github.com/liruizhe/CoSFiD.
4. The compilation-sensitive code files mentioned in Section 4.3 will be included in the supplement.

Acknowledgements

This work is supported in part by the Natural Science Foundation of China (no. 41275098), the National Grand Fundamental Research 973 Program of China (no. 2014CB441302) and the Tsinghua University Initiative Scientific Research Program (no. 20131089356).

References

Alexander, K., & and Easterbrook, S. M.: The software architecture of climate models: a graphical comparison of CMIP5 and EMICAR5 configurations. Geosci. Model Dev., 8, 1221-1232, 2015.

Baker, A. H., Hammerling, D. M., Levy, M. N., Xu, H., Dennis, J. M., Eaton, B. E., ... & Williamson, D.: A new ensemble-based consistency test for the Community Earth System Model. Geosci. Model Dev. Discuss., 8, 3823-3859, 2015.

Easterbrook, S. M. and Johns, T. C.: Engineering the software for understanding climate change, Comput. Sci. Eng., 11, 65–74, 2009.

Hong, S. Y., Koo, M. S., Jang, J., Esther Kim, J. E., Park, H., Joh, M. S., ... & Oh, T. J.: An Evaluation of the Software System Dependency of a Global Atmospheric model, Mon.Weather Rev., 141, 4165–4172, 2013.

Hurrell, J. W., Holland, M. M., Gent, P. R., Ghan, S., Kay, J. E., Kushner, P. J., ... & Marshall, S.: The community earth system model: a framework for collaborative research. Bulletin of the American Meteorological Society, 94(9), 1339-1360, 2013.

Li, L., Lin, P., Yu, Y., Wang, B., Zhou, T., Liu, L., ... & Qiao, F.: The flexible global ocean-atmosphere-land system model, Grid-point Version 2: FGOALS-g2. Advances in Atmospheric Sciences, 30, 543-560, 2013a.

Li, L., Wang, B., Dong, L., Liu, L., Shen, S., Hu, N., ... & Yang, G.: Evaluation of grid-point atmospheric model of IAP LASG version 2 (GAMIL2). Advances in Atmospheric Sciences, 30, 855-867, 2013b.

Liu, H. L., Zhang, X.H., Li, W., Yu, Y.Q., Yu, R.C.: A eddy-permitting oceanic general circulation model and its preliminary evaluations, Adv. Atmos. Sci. 21, 675–690, 2004.

Liu, L., Li, R., Zhang, C., Yang, G., Wang, B., and Dong, L.: Enhancement for bitwise identical reproducibility of Earth system modeling on the C-Coupler platform, Geosci. Model Dev. Discuss., 8, 2403-2435, doi:10.5194/gmdd-8-2403-2015, 2015a.

Liu, L., Peng, S., Zhang, C., Li, R., Wang, B., Sun, C., ... & Yang, G.: Importance of bitwise identical reproducibility in earth system modeling and status report, Geosci. Model Dev. Discuss., 8, 4375–4400, 2015b.

Liu, J.: Sensitivity of sea ice and ocean simulations to sea ice salinity in a coupled global climate model. Science China Earth Sciences, 53(6), 911-918, 2010.

Neale, R. B., Chen, C. C., Gettelman, A., Lauritzen, P. H., Park, S., Williamson, D. L., ... & Taylor, M. A.: Description of the NCAR community atmosphere model (CAM 5.0). NCAR Tech. Note NCAR/TN-486+ STR, 2010.

Oleson, K. W., Dai, Y., Bonan, G., Bosilovich, M., Dickinson, R., Dirmeyer, P., ... & Zeng, X.: Technical description of the community land model (CLM). NCAR Technical Note NCAR/TN-461+ STR, National Center for Atmospheric Research, Boulder, CO, 2004.

Smith, R., Jones, P., Briegleb, B., Bryan, F., Danabasoglu, G., Dennis, J., ... & Yeager, S.: The Parallel Ocean Program (POP) Reference Manual Ocean Component of the Community Climate System Model (CCSM) and Community Earth System Model (CESM). Rep. LAUR-01853, 141, 2010.

Song, Z., Qiao, F., Lei, X., and Wang, C.: Influence of parallel computational uncertainty on simulations of the Coupled General Climate Model, Geosci. Model Dev., 5, 313–319, doi:10.5194/gmd-5-313-2012, 2012.

Wang, X. C., Liu, J. P., Yu, Y. Q., Liu, H. L., & Li, L. J.: Numerical simulation of polar climate with FGOALS-g1. 1. Acta Meteorologica Sinica, 67(6), 961-972, 2009.

Table 1. Compiler families used for Earth system modeling. They are from the supported compiler lists of several ESMs.

Compiler family	Free or commercial	Supported hardware platforms	Supported programming languages
GNU	Free	Almost all common platforms	Fortran, C, C++, etc.
Intel	Commercial	x86 and x86-64bit architectures	Fortran, C, C++
PGI	Commerical	x86, x86-64bit, CUDA, and ARM architectures	Fortran, C, C++
Lahey	Commercial	x86 and x86-64bit architectures	Fortran
PathScale EKOPath	Commercial	x86 and x86-64bit architectures	Fortran, C, C++
Cray	Commercial	Cray supercomputer series (x86, x86-64bit and CUDA architectures)	Fortran, C, C++

1 Table 2. Five latest versions of the Intel compilers.

Compiler version	Release date
11.1	June 23, 2009
12.1	September 8, 2011
13.0	September 5, 2012
14.0.1	October 18, 2013
15.0.1	October 30, 2014

2

1 Table 3. Intel compiler optimization options that may impact the precision of floating-point
2 calculation. They are common to the compiler versions listed in Table 2.

Compiler optimization option	Description
-fp-model [fast|precise|strict] [source]	Controls the semantics of floating-point calculations: fast: Enables more aggressive optimizations on floating-point data. precise: Enables value-safe optimizations on floating-point data. strict: Enables precise and except, disables contractions, and enables pragma stdc fenv_access. Source: Rounds intermediate results to source-defined precision and enables value-safe optimizations.
-fp-speculation fast|safe|strict	Tells the compiler the mode in which to speculate on floating-point operations. fast: Tells the compiler to speculate on floating-point operations. safe: Tells the compiler to disable speculation if there is a possibility that the speculation may cause a floating-point exception. strict: Tells the compiler to disable speculation on floating-point operations.
-mp1	Improves floating-point precision and consistency.
-[no-]vec	Enables or disables vectorization.
-[no-]simd	Enables or disables the SIMD (Single instruction, multiple data) vectorization feature of the compiler.
-[no-]fp-port	Rounds floating-point results after floating-point operations.

-[no-]ftz	Flushes denormal results to zero.
-pc[n]	Enables control of floating-point significant precision.
-[no-]prec-div	Improves precision of floating-point divides.
-[no-]prec-sqrt	Improves precision of square root implementations.

1

Table 4. Five latest versions of the GCC compilers. The release date of a given compiler version in the table is the release date of its latest revision version.

Compiler version	Release date
4.6.4	April 12, 2013
4.7.4	April 13, 2013
4.8.5	June 23, 2015
4.9.3	June 26, 2015
5.1	April 22, 2015

Table 5. GCC compiler optimization options that may impact the precision of floating-point calculation. They are common to the compiler versions listed in Table 4.

Compiler flag	Description
-ffloat-store	Do not store floating-point variables in registers, and inhibit other options that might change whether a floating-point value is taken from a register or memory.
-ffast-math	Sets -fno-math-errno, -funsafe-math-optimizations, -ffinite-math-only, -fno-rounding-math, -fno-signaling-nans and -fcx-limited-range.
-f[no-]unsafe-math-optimizations	Allow optimizations for floating-point arithmetic that (a) assume that arguments and results are valid and (b) may violate IEEE or ANSI standards. When used at link-time, it may include libraries or startup files that change the default FPU control word or other similar optimizations.
-f[no-]associative-math	Allow re-association of operands in series of floating-point operations.
-f[no-]reciprocal-math	Allow the reciprocal of a value to be used instead of dividing by the value if this enables optimizations.
-f[no-]finite-math-only	Allow optimizations for floating-point arithmetic that assume that arguments and results are not NaNs or ±Infs.
-f[no-]rounding-math	Disable transformations and optimizations that assume default floating-point rounding behavior.
-f[no-]cx-limited-range	When enabled, this option states that a range reduction step is not needed when performing complex division. Also, there is no checking whether the result of a complex multiplication or division is "NaN + I*NaN", with an attempt to rescue the situation in that case.

1

Table 6. Intel compiler flags that are based on the compiler optimization options given in Table 3. The first compiler flag is the strictest one (which limits compiler optimizations most significantly), while every other compiler flag is derived from the first one through changing only one compiler optimization option.

No.	Compiler flag
1	-fp-model strict -fp-speculation=strict -mp1 -no-vec –no-simd
2	-fp-model precise -fp-speculation=strict -mp1 -no-vec –no-simd
3	-fp-model fast -fp-speculation=strict -mp1 -no-vec –no-simd
4	-fp-model source -fp-speculation=strict -mp1 -no-vec –no-simd
5	-fp-model strict -fp-speculation=safe -mp1 -no-vec –no-simd
6	-fp-model strict -fp-speculation=fast -mp1 -no-vec –no-simd
7	-fp-model strict -fp-speculation=strict -no-vec –no-simd
8	-fp-model strict -fp-speculation=strict -mp1 -vec –no-simd
9	-fp-model strict -fp-speculation=strict -mp1 -no-vec –simd

Table 7. GCC compiler flags that are based on the compiler optimization options listed in Table 5. The first compiler flag is the strictest one (which limits compiler optimizations most significantly), while every other compiler flag is derived from the first one through changing only one compiler optimization option.

No.	Compiler flag
1	-ffloat-store -fno-unsafe-math-optimizations -fno-associative-math -fno-reciprocal-math -fno-finite-math-only -fno-rounding-math -fno-cx-limited-range
2	-fno-unsafe-math-optimizations -fno-associative-math -fno-reciprocal-math -fno-finite-math-only -fno-rounding-math -fno-cx-limited-range
3	-ffloat-store -funsafe-math-optimizations -fno-associative-math -fno-reciprocal-math -fno-finite-math-only -fno-rounding-math -fno-cx-limited-range
4	-ffloat-store -fno-unsafe-math-optimizations -fassociative-math -fno-reciprocal-math -fno-finite-math-only -fno-rounding-math -fno-cx-limited-range
5	-ffloat-store -fno-unsafe-math-optimizations -fno-associative-math -freciprocal-math -fno-finite-math-only -fno-rounding-math -fno-cx-limited-range
6	-ffloat-store -fno-unsafe-math-optimizations -fno-associative-math -fno-reciprocal-math -ffinite-math-only -fno-rounding-math -fno-cx-limited-range
7	-ffloat-store -fno-unsafe-math-optimizations -fno-associative-math -fno-reciprocal-math -fno-finite-math-only -frounding-math -fno-cx-limited-range
8	-ffloat-store -fno-unsafe-math-optimizations -fno-associative-math -fno-reciprocal-math -fno-finite-math-only -fno-rounding-math -fcx-limited-range

Table 8: Simulation results of CAM5 with various compiling setups of Intel compilers. The compiler flags are given in Table 6. Each color represents a bitwise identical result except for the white. A simulation result that emerges only once is in white color with a unique number.

Compiler optimization level	No. of compiler flag	Version of Intel compiler				
		11	12	13	14	15
-O0	1					
	2					
	3	(1)				
	4					
	5					
	6					
	7					
	8					
	9					
-O1	1					
	2					
	3	(2)	(3)	(4)	(5)	(6)
	4					
	5					
	6					
	7					
	8					
	9					
-O2	1					
	2					
	3	(7)	(8)	(9)	(10)	(11)
	4					
	5					
	6					
	7					
	8					
	9					
-O3	1					
	2					
	3	(12)	(13)	(14)	(15)	(16)
	4					
	5					
	6					
	7					
	8					
	9					

Table 9: Similar to Table 8 except for the simulation results of POP2. Each table cell with "--" means that the compilation of POP2 fails under the corresponding compiling setup, due to issue DPD200178252 of Intel compilers (https://software.intel.com/en-us/articles/intel-composer-xe-2013-compilers-fixes-list).

Compiler optimization level	No. of compiler flag	Version of Intel compiler				
		11	12	13	14	15
-O0	1		--			
	2					
	3	(1)				
	4					
	5		--			
	6		--			
	7		--			
	8		--			
	9		--			
-O1	1		--			
	2					
	3	(2)			(3)	(4)
	4					
	5		--			
	6		--			
	7		--			
	8		--			
	9		--			
-O2	1		--			
	2					
	3	(5)			(6)	(7)
	4					
	5		--			
	6		--			
	7		--			
	8		--			
	9		--	(8)		
-O3	1		--			
	2					
	3	(9)	(10)	(11)	(12)	(13)
	4					
	5		--			
	6		--			
	7		--		xzazsxz	
	8		--	(14)		
	9		--	(15)		

1 Table 10: Similar to Table 8 except for the simulation results of FGOALS-g2.

Compiler optimization level	No. of compiler flag	Version of Intel compiler				
		11	12	13	14	15
-O0	1					
	2					
	3	(1)				
	4					
	5					
	6					
	7					
	8					
	9					
-O1	1					
	2					
	3	(2)	(3)	(4)	(5)	(6)
	4					
	5					
	6					
	7					
	8					
	9					
-O2	1					
	2					
	3	(7)	(8)	(9)	(10)	(11)
	4					
	5					
	6					
	7					
	8					
	9					
-O3	1					
	2					
	3	(12)	(13)	(14)	(15)	(16)
	4					
	5					
	6					
	7					
	8					
	9					

2

Table 11: Simulation results of CAM5 with various compiling setups of GCC compilers. The compiler flags are given in Table 7. Each color represents a bitwise identical result except the white. A simulation result that emerges only once is in white color with a unique number.

Compiler optimization level	No. of compiler flag	Version of GCC compiler				
		4.6.4	4.7.4	4.8.5	4.9.3	5.1.0
-O0	1					
	2					
	3	(1)				
	4					
	5	(2)				
	6					
	7					
	8	(3)				
-O1	1					
	2					
	3	(4)	(5)			(6)
	4					
	5	(7)	(8)			(9)
	6					
	7					
	8					
-O2	1					
	2					
	3	(10)	(11)			(12)
	4					
	5	(13)	(14)	(15)	(16)	(17)
	6					
	7					
	8					
-O3	1					
	2					
	3	(18)	(19)			(20)
	4					
	5	(21)	(22)	(23)	(24)	(25)
	6					
	7					
	8		(26)			

Table 12: Similar to Table 11 except for the simulation results of POP2.

Compiler optimization level	No. of compiler flag	Version of GCC compiler				
		4.6.4	4.7.4	4.8.5	4.9.3	5.1.0
-O0	1					
	2					
	3	(1)				
	4					
	5					
	6					
	7					
	8					
-O1	1					
	2					
	3					
	4					
	5					(2)
	6					
	7					
	8					
-O2	1					
	2					
	3					
	4					
	5					(3)
	6					
	7					
	8					
-O3	1					
	2					
	3	(4)	(5)	(6)	(7)	(8)
	4					
	5					(9)
	6					
	7					
	8					

Table 13: Similar to Table 11 except for the simulation results of FGOALS-g2. FGOALS-g2 has not been compiled using the GCC compilers for simulation runs before. Therefore a large proportion of simulation runs are failed (marked with "--" in the table). For example, crashes or deadlocks are encountered under compiler optimization levels O1 to O3.

Compiler optimization level	No. of compiler flag	Version of GCC compiler				
		4.6.4	4.7.4	4.8.5	4.9.3	5.1.0
-O0	1					
	2					
	3	(1)		(2)		
	4					
	5	(3)	(4)			
	6					
	7					
	8					
-O1	1	--	--	--	--	--
	2	--	--	--	--	--
	3	--	--	--	--	--
	4	--	--	--	--	--
	5	--	--	--	--	--
	6	--	--	--	--	--
	7	--	--	--	--	--
	8	--	--	--	--	--
-O2	1	--	--	--	--	--
	2	--	--	--	--	--
	3	--	--	--	--	--
	4	--	--	--	--	--
	5	--	--	--	--	--
	6	--	--	--	--	--
	7	--	--	--	--	--
	8	--	--	--	--	--
-O3	1	--	--	--	--	--
	2	--	--	--	--	--
	3	--	--	--	--	--
	4	--	--	--	--	--
	5	--	--	--	--	--
	6	--	--	--	--	--
	7	--	--	--	--	--
	8	--	--	--	--	--

Table 14: Ideal bitwise identical compiling setup sets of the three models when using Intel compilers. Each color except the white corresponds to an ideal bitwise compiling setup set.

Compiler optimization level	No. of compiler flag	Version of Intel compiler				
		11	12	13	14	15
-O0	1					
	2					
	3					
	4					
	5					
	6					
	7					
	8					
	9					
-O1	1					
	2					
	3					
	4					
	5					
	6					
	7					
	8					
	9					
-O2	1					
	2					
	3					
	4					
	5					
	6					
	7					
	8					
	9					
-O3	1					
	2					
	3					
	4					
	5					
	6					
	7					
	8					
	9					

Table 15: Examples of different results of the calculation at line 330 of Fig. 2 when changing the compiler optimization level from *O1* to *O2*. The input of the calculation is the same (bitwise identical) at both compiler optimization levels. The different digits in the results are highlighted in red.

Variables			Example		
			No. 1	No. 2	No. 3
Input	k		2	3	3
	i		9	1	5
	ipair		1	1	1
	dryvol_t_new(ipair,i,k)		1.245177471001780E-013	1.367964902074264E-013	1.362619492656580E-013
	num_t_oldbnd(ipair,i,k)		660367763.850537	673856916.583178	665467981.351062
	factoraa(mfrm)		1.41486733199200	1.41486733199200	1.41486733199200
Output	dgn_t_new(ipair,i,k)	optimization level O1	5.107909846347498E-008	5.235166698430766E-008	5.250216806732902E-008
		optimization level O2	5.107909846347492E-008	5.235166698430762E-008	5.250216806732898E-008

Table 16: Assembly codes of the calculation at line 330 in Fig. 2 in two compiler optimization levels (*O1* and *O2*). The most significant difference of the assembly codes is the calling of different power functions.

Optimization level O1	Optimization level O2
movq %rsi, -40(%rbp)	**movsd** %xmm7, -344(%rbp)
movq %r8, -32(%rbp)	**movsd** %xmm1, -320(%rbp)
movq %r9, -24(%rbp)	**movsd** %xmm3, -312(%rbp)
movsd %xmm8, -16(%rbp)	**movsd** %xmm2, -304(%rbp)
call pow	**call** cbrt

Table 17: An example of obvious different results in lines 1923-1932 of Fig. 3 when changing the compiler optimization levels (from O2 to O3). A manual result calculated by Python is also provided.

Variables			Value
Input	WORK3(i, j)		0.00000000000000
	dz(k)		1000.00000000000
	KAPPA_ISOP(i, j, kbt, k, bid)		1.991793396882581E-006
	SLX(i, j, ieast, kbt, k, bid)		-0.120114394362605
	HYX(i, j, bid)		1.32933181234324
	TX(i, j, k, n, bid)		-0.161989629268646
	SLY(i, j, jnorth, kbt, k, bid)		-0.13980933 0009777
	HXY(i, j, bid)		0.761427260631393
	TY(i, j, k, n, bid)		0.152039408683777
	SLX(i, j, iwest, kbt, k, bid)		-7.344871974748127E-002
	HYX(i-1, j, bid)		1.32933181234324
	TX(i-1, j, k, n, bid)		-0.192202091217041
	SLY(i, j, jsouth, kbt, k, bid)		-0.112685940441552
	TY(i, j-1, k, n, bid)		0.743088001419362
	TY(i, j-1, k, n, bid)		0.122525990009308
Output	WORK3(i, j)	Execution result (at optimization level O2)	3.622331248054413E-005
		Execution result (at optimization level O3)	3.571897176404182E-005
		Manual result (by Python)	3.62233124805436E-05

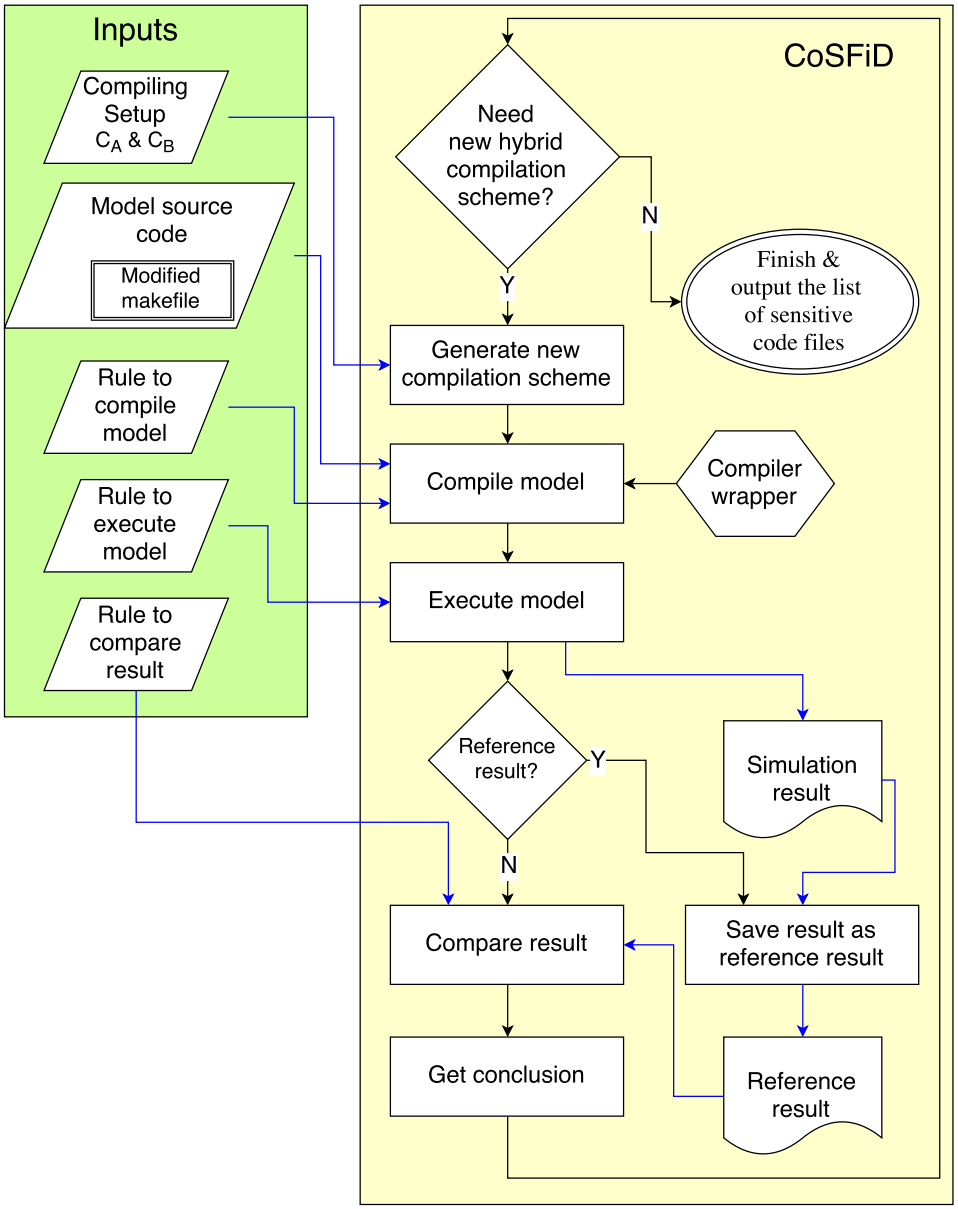

Figure 1: Flowchart of CoSFiD for detecting compilation-sensitive code files. In each iteration, CoSFiD first checks whether it is necessary to generate a new hybrid compilation scheme (some code files are compiled with C_A and the remaining code files are compiled with C_B). If unnecessary, which means the whole process of the detection should end, CoSFiD will output all compilation-sensitive code files. Otherwise, CoSFiD generates a new hybrid compilation scheme, and then calls the corresponding rule to compile the model code using the compiler wrapper and run the simulation. If it is the first run of the simulation, which also means all code files are compiled with C_A, the simulation result will be recorded as the reference result. Otherwise, CoSFiD calls the corresponding rule to compare the simulation result to the reference result and then uses the conclusion to drive the next iteration.

```
321  !   num_t_old is total number in particles/kmol-air
322          num_t_old = q(i,k,numptr_amode(mfrm)-loffset)
323          num_t_old = num_t_old + qqcw(i,k,numptrcw_amode(mfrm)-loffset)
324          num_t_old = max( 0.0_r8, num_t_old )
325          dryvol_t_oldbnd = max( dryvol_t_old, dryvol_smallest(mfrm) )
326          num_t_oldbnd = min( dryvol_t_oldbnd*v2nlorlx(mfrm), num_t_old )
327          num_t_oldbnd = max( dryvol_t_oldbnd*v2nhirlx(mfrm), num_t_oldbnd )
328
329  !   no renaming if dgnum < "base" dgnum,
330          dgn_t_new = (dryvol_t_new/(num_t_oldbnd*factoraa(mfrm)))**onethird
331          if (dgn_t_new .le. dgnum_amode(mfrm)) cycle mainloop1_ipair
332
333  !   compute new fraction of number and mass in the tail (dp > dp_cut)
334          lndgn_new = log( dgn_t_new )
335          lndgv_new = lndgn_new + dum3alnsg2(ipair)
336          yn_tail = (lndp_cut(ipair) - lndgn_new)*factoryy(mfrm)
337          yv_tail = (lndp_cut(ipair) - lndgv_new)*factoryy(mfrm)
338          tailfr_numnew = 0.5_r8*erfc( yn_tail )
339          tailfr_volnew = 0.5_r8*erfc( yv_tail )
```

Figure 2: Part of the code lines of the compilation-sensitive code segment in the code file *modal_aero_rename.F90* of CAM5. It is found that the code at line 330 can produce different results when different compiling setups are used.

```
1920              do j=this_block%jb,this_block%je
1921                do i=this_block%ib,this_block%ie
1922
1923                   WORK3(i,j) = WORK3(i,j)                                  &
1924                       + ( dz(k) * KAPPA_ISOP(i,j,kbt,k,bid)                &
1925                       * ( SLX(i,j,ieast ,kbt,k  ,bid)                      &
1926                          * HYX(i  ,j  ,bid) * TX(i  ,j  ,k  ,n,bid)        &
1927                         + SLY(i,j,jnorth,kbt,k  ,bid)                      &
1928                          * HXY(i  ,j  ,bid) * TY(i  ,j  ,k  ,n,bid)        &
1929                         + SLX(i,j,iwest ,kbt,k  ,bid)                      &
1930                          * HYX(i-1,j  ,bid) * TX(i-1,j  ,k  ,n,bid)        &
1931                         + SLY(i,j,jsouth,kbt,k  ,bid)                      &
1932                          * HXY(i  ,j-1,bid) * TY(i  ,j-1,k  ,n,bid) ) )
1933
1934                enddo
1935              enddo
1936
1937              do j=this_block%jb,this_block%je
1938                do i=this_block%ib,this_block%ie
1939
1940                   WORK3(i,j) = WORK3(i,j)                                  &
1941                       + ( SF_SLX(i  ,j  ,ieast ,kbt,k  ,bid)               &
1942                          * HYX(i  ,j  ,bid) * TX(i  ,j  ,k  ,n,bid)        &
1943                         + SF_SLY(i  ,j  ,jnorth,kbt,k  ,bid)               &
1944                          * HXY(i  ,j  ,bid) * TY(i  ,j  ,k  ,n,bid)        &
1945                         + SF_SLX(i  ,j  ,iwest ,kbt,k  ,bid)               &
1946                          * HYX(i-1,j  ,bid) * TX(i-1,j  ,k  ,n,bid)        &
1947                         + SF_SLY(i  ,j  ,jsouth,kbt,k  ,bid)               &
1948                          * HXY(i  ,j-1,bid) * TY(i  ,j-1,k  ,n,bid) )
1949
1950                enddo
1951              enddo
```

Figure 3: Part of the code lines of the compilation-sensitive code segment in the code file *hmix_gm.F90* of POP2. It is found that the code from line 1923 to line 1932 can produce significantly different results when different compiling setups are used.

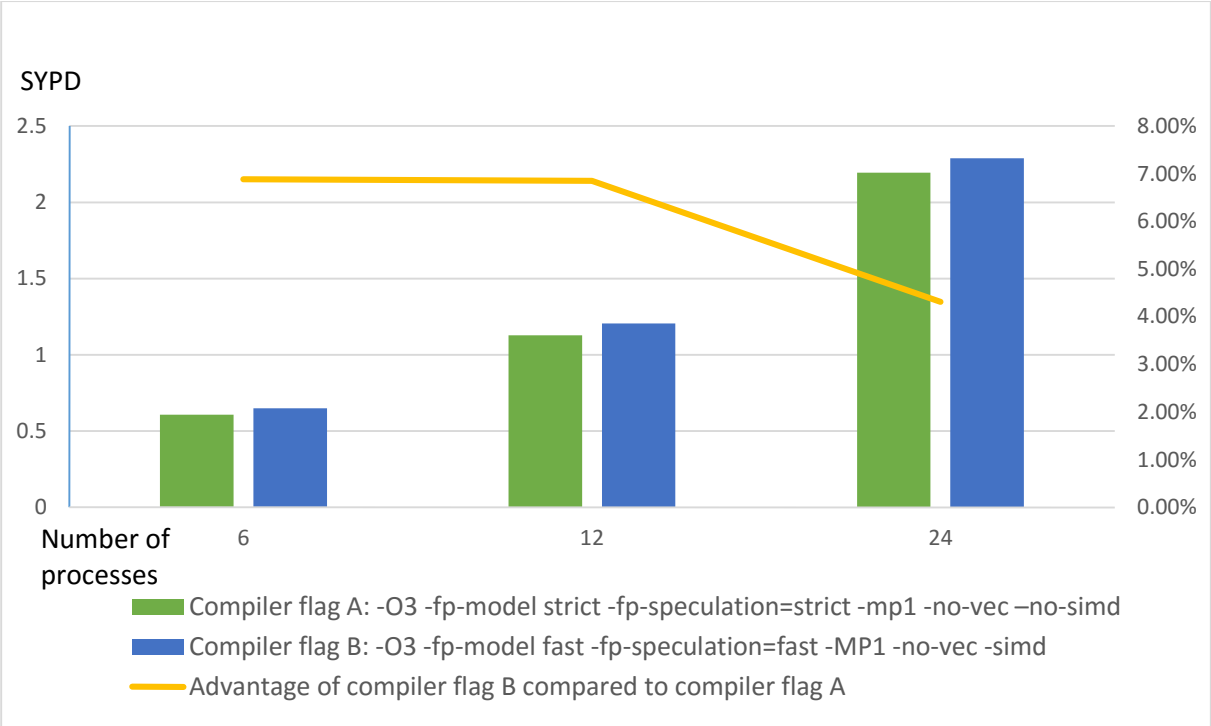

Figure 4: Simulation speed (simulated years per day; SYPD) of CAM5 under two compiler flags (*A* and *B*) of Intel compiler version 13 when increasing the number of processes from 6 to 24. The high-performance computer Tansuo100 is used for this test. Compiler flag *A* ("*-O3 -fp-model strict -fp-speculation=strict -mp1 -no-vec –no-simd*") is from the biggest bitwise identical compiling setup sets in Table 8. Compiler flag *B* ("*-O3 -fp-model fast -fp-speculation=fast -MP1 -no-vec -simd*") should be the compiler flag for fastest simulation speed. Compiler flag "*-O3 -fp-model fast -fp-speculation=fast -MP1 -vec -simd*" should be more aggressive than compiler flag *B* in compiler optimizations. It is not used in this test because the corresponding simulation run of CAM5 crashes. The advantage of compiler flag B compared to compiler flag A is defined as the performance improvement when compiler flag is changed from A to B.

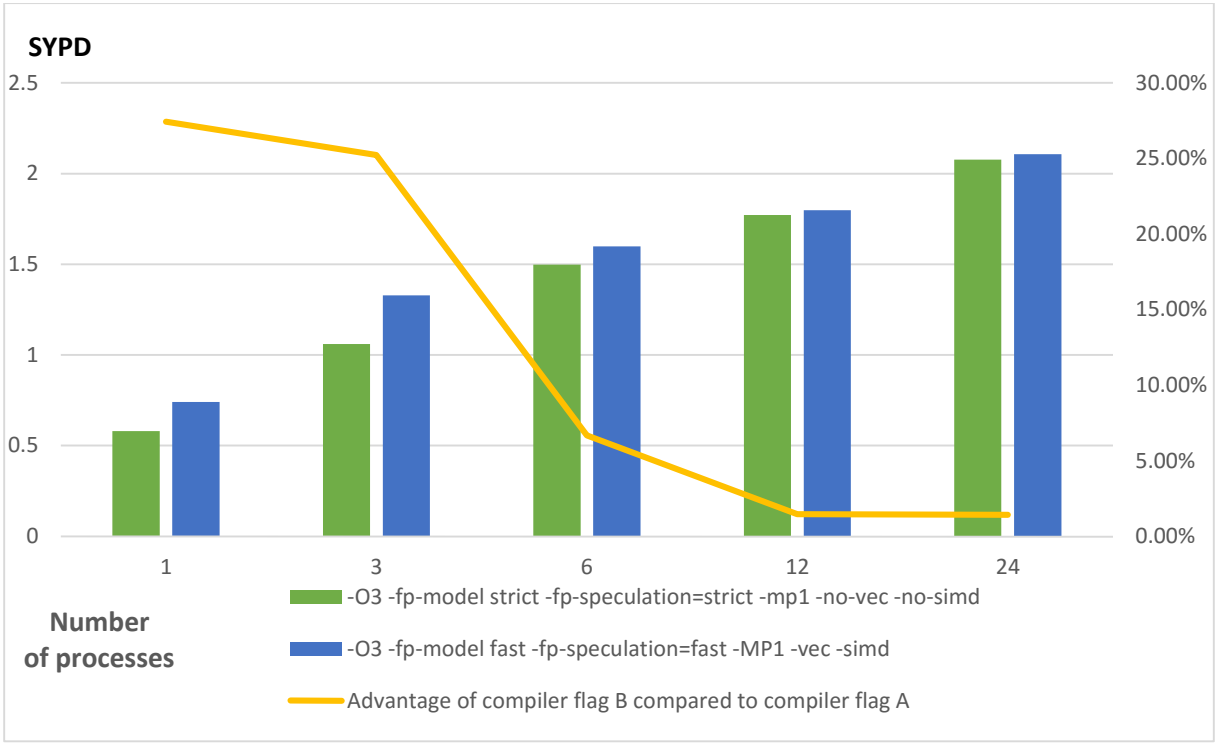

Figure 5: Simulation speed (simulated years per day; SYPD) of GAMIL2 (Li et al, 2013b) under two compiler flags (*A* and *B*) of Intel compiler version 13 when increasing the number of processes from 1 to 24. The high-performance computer Tansuo100 is used for this test. Compiler flag *A* ("*-O3 -fp-model strict -fp-speculation=strict -mp1 -no-vec -no-simd*") is the also the compiler flag *A* used in Fig. 4. Compiler flag *B* ("*-O3 -fp-model fast -fp-speculation=fast -MP1 -vec -simd*") should be the compiler flag for fastest simulation speed. The advantage of compiler flag B compared to compiler flag A is defined as the performance improvement when compiler flag is changed from A to B.