iEMSs International Congress on Environmental Modelling and Software

9th International Congress on Environmental Modelling and Software
Fort Collins, Colorado, USA, Mazdak Arabi, Olaf David, Jack Carlson, Daniel P. Ames (Eds.)
https://scholarsarchive.byu.edu/iemssconference/2018/

Application Software Analytics Toolkit for Facilitating the Understanding, Componentization, and Refactoring of Large-Scale Scientific Models

Dali Wang[1], Fengguang Song[2], Weijian Zheng[2]
[1] Oak Ridge National Laboratory, Oak Ridge, TN 37831. USA. wangd@ornl.gov
[2] Indiana University-Purdue University Indianapolis, IN 46212. {song412, zheng273}@purdue.edu

Abstract: The complexity of large scientific models developed under certain machine architectures and application requirements assumptions has become a real barrier that impedes continuous software development, including adding new features and functions, validating domain knowledge incorporated in the software systems, offering portable high performance, as well as redesigning and refactoring code for emerging computational platforms. In this study, we leverage experience from several practices, including open-source software engineering research, software dependency understanding, compiler technologies, analytical performance modeling, micro-benchmarks, and functional unit testing, to design software toolkit to better understand and enhance software productivity and performance. Our software tools are designed to collect the information of scientific codes and extract the common features of legacy codes. In this work, we will focus on the front-end of our system (Software X-ray Scanner): a metric information collection system for better understanding of key scientific functions and associated software dependency. We use several science codes from the Innovative and Novel Computational Impact on Theory and Experiment (INCITE) program and Exascale Computing Projects (ECPs), Subsurface Biogeochemical Research Models to explore and recommend cost-efficient approaches for program understanding and code refactoring. The toolkits will increase the software productivity for the Interoperable Design of Extreme-scale Application Software (IDEAS) community which is supported by both US Department of Energy's Advanced Scientific Computing Research (ASCR) and Biological and Environmental Research (BER) programs. We also expect that these toolkits can benefit broader scientific communities that are facing similar challenges.

***Keywords**:* application software analysis, high performance application, program understanding and refactoring, software X-ray

1 INTRODUCTION, BACKGROUND and MOTIVATION

The complexity of large scientific models developed under certain machine architectures and application requirements assumptions has become a real barrier that impedes continuous software development, including adding new features and functions, validating domain knowledge incorporated in the software systems, offering portable high performance. In this study, we leverage experience from several practices to design software toolkits to better understand and enhance software productivity and performance of large-scale scientific codes. We will design software tools that contain two systems: 1) a Software X-ray Scanner: a metric information collection system for better understanding of key scientific functions and associated software/library dependency, and 2) a software data analyzer: a system to facilitate the integration and refactoring of key scientific functions and modules. In this paper we only focus on the design considerations of the Software X-ray Scanner. We use several science codes from the Innovative and Novel Computational Impact on Theory and Experiment (INCITE) program and Exascale Computing Projects (ECPs), Subsurface Biogeochemical Research Models to explore and recommend cost-efficient approaches for program understanding and code refactoring.

2 APPROACHES TO DESIGN A SOFTWARE X-RAY SCANNER

2.1 HPC Software Structure and Function Analysis

To understand the internal structure or the software architecture of an HPC software package, we first use static software analysis tools to analyze function compositions and construct the relationship among functions. The collected function-level information will help users increase their understanding of the software. For instance, various software tools can extract call graphs from the source code. Unlike software engineering tools which often target debug, security, and potential runtime errors, the goal of this project is to present information that can be easily consumed by humans.

In addition to collecting the function-level information, the Software X-ray Scanner can extract and collect the high-level software and hardware-relevant information from the open source software package. The Software X-ray Scanner goes through two steps to fulfill the goals. *In the first step*, it collects the information of programming languages, parallel programming models, compiler options, dependent third-party libraries, and required external projects. To get the information, we design and implement a dedicated python function to parse and process each CMake command that may exist in CMakeLists (see more details in Section 2.2). A CMake Parser is used in the implementation. Since AutoConf macros have one-to-one relationship with CMake, the command-specific python function is also extended to parse AutoConf macros. Moreover, the third-party libraries are shown in a directed acyclic graph (DAG) to show their dependencies. *In the second step*, we use a compiler plugin to analyze the source code level information to extract more detailed information. For instance, the scanner can search for specific MPI-2 requirements, OpenMP specification, FPGA interface, AVX2.0 or AVX512, and so on. These hardware and software features will be printed out and presented in a table and a graph correspondingly. Hence, this work is able to to automatically identify architecture-dependent features that exist or are hidden in a software package but may not be portable to other computer systems. Instead of building a domain-specific tool, we design a generic toolkit to perform software analysis on generic HPC software packages.

The toolkit can collect the information of the source code, analyze the library dependencies, reveal special software and hardware features used by the code, as well as identifying requirement of special tools and specific compiler versions. For instance, certain open source HPC software package (such as INCITE applications and ECP applications) critically rely on GPU, FPGA, MIC, burst buffers, SSE/AVX, and new programming models (i.e., not using MPI) to deliver scalable high performance. Eventually, our tool works like an “x-ray” scanner, which can scan any software package and construct the software anatomy. Based on the software anatomy, users may easily get the “whole” picture of software functionality and hardware functionality (including the HPC features). Moreover, users can quickly decide which software package is more suitable to work/port on a different HPC system. Python tools and compiler plugins will be designed and developed to achieve the goal.

2.2 Dependency Analysis via Common Software Build Systems on HPC Systems

In this section, we briefly introduce the widely used GNU Build System [7] and the CMake Build System [12], which conveniently control the process of software compilation, library dependency checking, software/hardware/architecture checking, and third-party library linking.

First, the GNU Build System, also known as AutoTools, is used on many Unix-like computer systems. It was firstly introduced in 1995 and since then has been adopted by many free software and open source packages [4]. Autotools consists of utility programs of *AutoConf* [10] and *Automake* [11]. It works as a two-step process: 1) configure followed by 2) make. Given a *configure.ac* template file, running the command *autoconf* creates a *configure* script. The configure.ac template file is written in the form of GNU M4 [16] macros, and prepared to test the software and hardware system features a software package needs or will use. When executed, the generated *configure* script, will probe computer systems to test relevant features and convert the Makefile.in input file to the most commonly used Makefile. Finally, the *make* program reads the Makefile to create executable programs from source code. The *Makefile.in* input file can be either written by hand, or generated by the *automake* tool through writing a short Makefile.am file.

Second, the CMake Build System (or CMake) manages the software build process in an operating system independent and compiler-independent way. Different from AutoConf, CMake supports a wide variety of platforms including Windows, Mac OS, QNX, CYGWIN, and Android as well as most Unix-like platforms. It can generate native makefiles and workspaces (such as Visual Studio and Apple’s Xcode IDE) that can be used in various compiler environments of a user’s choice. The CMake building process is controlled by a number of *CMakeLists.txt* files under each source code subdirectory. Running

cmake will automatically generate building scripts based on the files of CMakeLists.txt. For instance, the building script on Unix is a set of Makefiles.

Both AutoConf and CMake allow software authors or developers to define various programming language features, compiler options, software dependencies, third-party and system libraries, hardware and architecture features, in configure.ac and CMakeLists.txt, respectively. Although CMake and AutoConf are distinct systems, their basic operations are the same although calling different functions. Most macros in AutoConf have corresponding commands in CMake. To list a few examples, AC_ARG_WITH in Autoconf is the same as the *option* command in cmake, AC_CHECK_LIB is the same as Check_Library_Exists, and etc...

3. HPC APPLICATIONS IN OUR EXPERIMENTS

We apply the Software X-ray Scanner toolkit to four exemplar scientific computing software packages: 1) E3SM: A global climate model that can simulate the Earth's past, present, and future climate states [1]; 2) QMCPACK: A many-body ab initio Quantum Monte Carlo code for computing the electronic structure of atoms, molecules, and solids [8]; 3) ParFlow: A numerical model that simulates the 3D groundwater flow, overland flow, and plant processes in complex real-world systems [13];and 4) ExaAM: An exascale simulation project to accelerate Additive Manufacturing (also known as 3D printing) [17]. These four applications use the Autoconf, CMake, or a hybrid of Autoconf and Cmake build systems, respectively. They also depend on a number of third-party libraries and external projects, and use MPI, OpenMP, CUDA, and parallel I/O, respectively.

4 PRELIMINARY OUTPUT FROM THE SOFTWARE X-RAY SCANNER

The high-level information extracted from the X-ray scanner is listed as follows:

- All third-party library components and composition: shown in a dependency graph, each with a required minimum version number.
- Computer architecture components: Does the software package require GPU, AVX, NUMA control, FPGA, parallel file system, burst buffer, NVLink, GPUDirect, etc... Based on the software building process configuration options, we also classify each of the hardware components into three categories: (i) Must have, (ii) performance critical, and (iii) able to run but may be slower without it.
- Communication layer: The software package uses an MPI library, RDMA, socket, or other special communication libraries.
- Programming model recognition: MPI, hybrid MPI/Pthreads/OpenMP, PGAS, AMT (asynchronous many tasks), or other parallel computing models.
- Programming languages: what specific languages are used and the minimum language version.
- Compilers: what compilers and versions are required by the software package.

5 RELATED WORK

Low level static software analysis tools are designed to analyze the source code to collect informations such as memory access violations, security flaws, functional dependencies and program errors, instead of high-level information such as software composition, library dependency, hardware features, specific compiler version, special tools. Take a few examples. Misha [25] compares a number of different software analysis tools to find the security flaws. An analysis tool called Archer [18] is designed to detect the memory access violation in the source code of C. It builds a calling graph of examination functions by parsing the source code. Dor et al. [5] build a tool to find the string errors in C code that may be exploited by computer viruses. Other low level static code analysis tools also aim at exploring dependencies among functions. For instance, Norman et al. [21] propose a tool for C language to extract definition dependency, calling dependency, functional and data flow dependencies in the source code. Bush et al. [3] create a tool for detecting possible program errors in C and C++ code by drawing the execution path. In addition, tools like Doxygen [19] also can be used to generate the code structure and document for different languages.

On the other hand, higher-level static analysis tools focus more on providing users with a high-level picture of the software. Wilhelm et al. [22] analyze Java packages and visualize the package design quality. The ScanCode toolkit [14] is developed to extract the license, copyright, dependency and other information from the source code. Similarly, fossology [6] also can provide user the license and copyright information. OSS Review toolkit [15] is an open source project to give user an insight into the dependencies of different open source libraries. They accomplish this task by incorporating other

package managers (e.g., MAVEN, PIP, NPM) and code scanners (e.g., Licenseem, ScanCode). Different from these toolkits, we focus on extracting information not only related to libraries, software features, but also hardware features and performance portability (e.g., GPU requirement, MPI-2 requirements, OpenMP specification, FPGA interface)

There are also software tools that support dynamic software analysis. Zirkelbach et al. [24] conduct the dynamic software analysis of a Perl-based software. They use Kieker [20] and Gelphi [2] as their analysis and visualization tool. Vampir [9] is an analysis tool that supports both static and dynamic software analysis. It can be used to find and solve the performance bottleneck. Its input is not limited to source code. Wu et al. [23] use run-time traces to investigate the dependencies between different programs.

ACKNOWLEDGMENTS

This research was funded by the U.S. Department of Energy (DOE), Office of Science, Advanced Scientific Computing Research (ASCR) (Interoperable Design of Extreme-scale Application Software (IDEAS)).

REFERENCES

[1] Bader, D., Collins, W., Jacob, R., Jones, P., Rasch, P., Taylor, M., ... & Williams, D. "Accelerated climate modeling for energy (ACME) project strategy and initial implementation plan." (2014).
[2] Bastian, Mathieu, Sebastien Heymann, and Mathieu Jacomy. "Gephi: an open source software for exploring and manipulating networks." Icwsm 8 (2009): 361-362.
[3] Bush, William R., Jonathan D. Pincus, and David J. Sielaff. "A static analyzer for finding dynamic programming errors." Software-Practice and Experience 30.7 (2000): 775-802.
[4] Calcote, John. Autotools: A Practitioner's Guide to GNU Autoconf, Automake, and Libtool. No Starch Press, 2010.
[5] Dor, Nurit, Michael Rodeh, and Mooly Sagiv. "CSSV: Towards a realistic tool for statically detecting all buffer overflows in C." ACM Sigplan Notices. Vol. 38. No. 5. ACM, 2003.
[6] Gobeille, Robert. "The fossology project." Proceedings of the 2008 international working conference on Mining software repositories. ACM, 2008.
[7] Gough, Brian. GNU scientific library reference manual. Network Theory Ltd., 2009.
[8] Kim, J., Esler, K., McMinis, J., Clark, B., Gergely, J., Chiesa, S.,. "QMCPACK simulation suite." (2014).
[9] Knüpfer, A., Brunst, H., Doleschal, J., Jurenz, M., Lieber, M., & Nagel, W. E. "The vampir performance analysis tool-set." Tools for High Performance Computing. Springer, Berlin, Heidelberg, 2008. 139-155.
[10] MacKenzie, David, Roland McGrath, and Noah Friedman. "Autoconf: Generating automatic configuration scripts." (1994).
[11] MacKenzie, David, Tom Tromey, and Alexandre Duret-Lutz. "GNU Automake." User Manual, for Automake version 1 (1995).
[12] Martin, Ken, and Bill Hoffman. Mastering CMake: a cross-platform build system. Kitware, 2010.
[13] Maxwell, R. M., Kollet, S. J., Smith, S. G., Woodward, C. S., Falgout, R. D., Ferguson, I. M., ... & Ashby, S. "ParFlow user's manual." International Ground Water Modeling Center Report GWMI1.2009 (2009): 129.
[14] Ombredanne, Philippe et. al, scancode-toolkit, (2016), GitHub, https://github.com/nexB/scancode-toolkit
[15] Schuberth, Sebastian et. al, oss-review-toolkit, (2017), at https://github.com/heremaps/oss-review-toolkit.
[16] Seindal, René. "GNU m4, version 1.4." Free Software Foundation 59 (1997).
[17] Turner, John et. al, oss-review-toolkit, (2017), GitHub repository, https://github.com/ExascaleAM
[18] Xie, Yichen, Andy Chou, and Dawson Engler. "Archer: using symbolic, path-sensitive analysis to detect memory access errors." ACM SIGSOFT Software Engineering Notes 28.5 (2003): 327-336.
[19] Van Heesch, Dimitri. "Doxygen: Source code documentation generator tool." http://www.doxygen.org, 2008.
[20] Van Hoorn, André, Jan Waller, and Wilhelm Hasselbring. "Kieker: A framework for application performance monitoring and dynamic software analysis." Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering. ACM, 2012.
[21] Wilde, Norman, Ross Huitt, and Scott Huitt. "Dependency analysis tools: reusable components for software maintenance." Software Maintenance, 1989., Proceedings., Conference on. IEEE, 1989.
[22] Wilhelm, Michael, and Stephan Diehl. "Dependency viewer-a tool for visualizing package design quality metrics." Visualizing Software for Understanding and Analysis, 2005. VISSOFT 2005. 3rd IEEE International Workshop on. IEEE, 2005.
[23] Wu, Yongzheng, Roland HC Yap, and Rajiv Ramnath. "Comprehending module dependencies and sharing." Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2. ACM, 2010.
[24] Zirkelbach, Christian, Wilhelm Hasselbring, and Leslie Carr. "Combining Kieker with Gephi for Performance Analysis and Interactive Trace Visualization." (2015): 26-28.
[25] Zitser, Misha. Securing software: An evaluation of static source code analyzers. Diss. Massachusetts Institute of Technology, 2003.

Do not use the numeral for the references section.

References should include (in the following order): Author Name(s), Initials, Date, Title of article with first letter uppercase, full Journal name, Volume (Number), page range. The page range must be hyphenated. A 4 mm indentation must be left for each reference. Examples are given below

Castronova, T.A., Goodall, J.L., 2010. A generic approach for developing process-level hydrologic modeling components. Environ. Modell. Softw. 25, 819-825.