International Congress on Environmental Modelling and Software

9th International Congress on Environmental Modelling and Software - Ft. Collins, Colorado, USA - June 2018

Jun 25th, 9:00 AM - 10:20 AM

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

Dali Wang
Oak Ridge National Laboratory, wangd@ornl.gov

fengguang song
Indiana University Purdue University Indianapolis, song412@purdue.edu

weijian zheng,
Indiana University Purdue University Indianapolis, zheng273@purdue.edu

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 for certain machine architectures and application requirements has become a real barrier that impedes continuous software development. In this study, we use experience from several practices, including open-source software engineering, software dependency understanding, compiler technologies, analytical performance modeling, micro-benchmarks, and functional unit testing, to design software toolkits to enhance software productivity and performance. Our software tools collect the information on scientific codes and extract the common features of these codes. In this paper, we 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 dependency. We use several science codes from the Innovative and Novel Computational Impact on Theory and Experiment (INCITE) program, Exascale Computing Projects (ECPs), Subsurface Biogeochemical Research (SBR) to explore cost-efficient approaches for program understanding and code refactoring. The toolkits 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 expect that these toolkits can benefit broader scientific communities that are facing similar challenges.

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

1 INTRODUCTION, BACKGROUND AND MOTIVATION

The complexity of large-scale scientific models developed for certain machine architectures and application requirements has become a significant barrier that impedes continuous software development, which include adding new feature and functions, validating domain knowledge incorporated in the software systems, offering portable high performance, as well as refactoring code for emerging computational platforms. In this study, we use experience from several practices to design software toolkits to enhance software productivity and performance of large-scale scientific codes. We are in the process of developing software that contains 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. This paper focuses on the design considerations and preliminary results of the Software X-ray Scanner. We use several science codes from the Innovative and Novel Computational Impact on Theory and Experiment (INCITE) program, Exascale Computing Projects (ECPs), and Subsurface Biogeochemical Research (SBR) to explore cost-efficient approaches for program understanding and code refactoring.

2 APPROACHES TO DESIGNING A SOFTWARE X-RAY SCANNER

2.1 Related Software Analysis Tools

A number of low-level static software analysis tools have been developed to analyze source code and to collect information such as memory access violations, security flaws, functional dependencies and program errors. These tools most often ignore high-level information such as software composition, library dependency, hardware features, specific compiler version, and special tools. For instance, Zitser compared various software analysis tools to find security flaws. An analysis tool called Archer (Xie et al. 2003) was designed to detect memory access violation in C source code. It built a calling graph of target functions by parsing the source code. Dor et al. built a tool to find the string errors in C code that may be exploited by computer viruses. A few low-level static code analysis tools also aim at exploring dependencies among functions. For instance, Wilde et al. proposed a C language tool to extract definition dependency, calling dependency, functional and data flow dependencies in the source code. Bush et al. created a tool to detect possible program errors in C and C++ code by drawing the execution path. In addition, tools like Doxygen (Van Heesch 2008) can generate the code structure and document for different languages.

On the other hand, higher-level static analysis tools typically focus on providing users with a high-level picture of the software. Wilhelm et al. analyzed Java packages and visualized the package design quality. The ScanCode toolkit (Ombredanne et al. 2016) was developed to extract the license, copyright, dependency and other information from the source code. Similarly, Fossology (Gobeille 2008) can provide users with the license and copyright information. Open Source Software oss-review-toolkit (Schuberth et al. 2017) is an open source project to give user an insight into the dependencies of different open source libraries. They accomplished this task by incorporating other package managers (e.g., MAVEN, PIP, NPM) and code scanners (e.g., Licenseem, ScanCode). There is also a class of software tools that support dynamic software analysis. Zirkelbach et al. conducted the dynamic software analysis on Perl-based software. They used Kieker (Van Hoorn et al. 2012) and Gephi (Bastian et al. 2009) for data analysis and visualization. Vampir (Knupffer et al. 2008) is an analysis tool that supports both static and dynamic software analysis. It can be used to find performance bottlenecks. Wu et al. used run-time traces to investigate the dependencies between different programs.

In this paper, we focus on extracting software information related to not only libraries, software features, but also hardware features and performance portability (e.g., GPU requirement, MPI-2 requirements, OpenMP specification, FPGA interface) from high-performance computing (HPC) systems.

2.2 Common Software Build Systems on HPC Systems

To understand the proposed analysis approach, we briefly introduce the widely used GNU Build System (Gough 2007) and the CMake Build System (Martin et al. 2010), which conveniently control the process of software compilation, library dependency checking, software/hardware/architecture checking, and third-party library linking.

The GNU Build System, also known as *AutoTools*, is used on many Unix-like computer systems. It was introduced in 1995 and since then has been adopted by many free software and open source packages (Calcote 2010). *Autotools* consists of utility programs of *AutoConf* (MacKenzie et al. 1994) and *Automake* (MacKenzie et al. 1995). It works as a two-step process: *configure* followed by *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 (Seindal 1997) macros, and is 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.

Differently, the CMake Build System (or *CMake*) manages the software build process in an operating system independent and compiler-independent way. Unlike *AutoConf*, *CMake* supports a much wider variety of platforms including Windows, Mac OS, QNX, CYGWIN, and Android as well as most Unix-like platforms. CMake 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 the cmake command will automatically generate building scripts based on the files of CMakeLists.txt. For instance, the building script on Unix will be 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 the configure.ac and CMakeLists.txt, respectively. Although *CMake* and *AutoConf* are distinct systems, their basic operations share some similarity. Most macros in *AutoConf* have corresponding commands in CMake. For example, AC_ARG_WITH in *Autoconf* is the same as the *option* command in *CMake*, and AC_CHECK_LIB is the same as Check_Library_Exists.

2.3 HPC Software Structure and Function Analysis

To understand the internal structure or the software architecture of an HPC software project, we first utilize static software analysis tools to analyze function compositions and construct the relationship among functions. The collected function-level information can increase users' understanding of the software. Also, various software tools can extract call graphs from the source code. Unlike conventional software engineering tools which target debug, security, and potential runtime errors, the objective of this work is to present information that can be easily consumed by users for domain-related analysis.

In addition to collecting the function-level information, the introduced Software X-ray Scanner can extract and collect 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 goal. *In the first step*, it collects the information of programming languages, parallel programming models, compiler options, dependent third-party libraries, and required external software projects. To get this type of information, we design and implement a Python utility to parse and process each *CMake* command that may exist in CMakeLists. A *CMake* Parser is used in the scanner implementation. Since *AutoConf* macros have one-to-one relationship with *CMake*, the command-specific Python utility can be easily extended to parse *AutoConf* macros. Moreover, we use a directed acyclic graph (DAG) to show the dependencies among third-party libraries. *In the second step*, we use a compiler plugin (e.g., LLVM Clang plugin) to analyze the source code and to extract more detailed code level and hardware-dependent information. For instance, the scanner can search for specific MPI-2 requirements, OpenMP specification, FPGA interface, AVX2.0 or AVX512 requirement, and so on. These hardware and software features are collected and visualized by the toolkit. Therefore, our toolkit is able to automatically identify architecture-dependent features that are embedded in a software package but may not be portable to other computer systems. The goal is to design a general-purpose toolkit to perform software analysis of various HPC software packages, instead of a domain-specific one.

This new toolkit is able to collect the information of the source code, analyze the library dependencies, reveal special software and hardware features used by the code, as well as to identify requirement of special tools and specific compiler versions. 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, this tool is working 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 are being designed and developed to achieve the goal.

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 simulate the Earth's past, present, and future scenarios (Bader 2014); 2) QMCPACK: A many-body *ab initio* Quantum monte carlo code for computing the electronic structure of atoms, molecules, and solids (Kim et al. 2014); 3) ParFlow: A numerical model that simulates the 3D groundwater flow, overland flow, and plant processes in complex real-world systems (Maxwell et al. 2009); and 4) ExaAM: An exascale simulation project to accelerate additive manufacturing (also known as 3D printing) (Turner et al. 2017). These four applications use the *Autoconf*, *CMake*, even a hybrid build system. They depend on third-party libraries and external projects, and use miscellaneous HPC technologies, such as MPI, OpenMP, CUDA, OpenACC, and parallel I/O.

4 PRELIMINARY OUTPUT FROM THE SOFTWARE X-RAY SCANNER

The information that can be extracted from the Software X-ray Scanner is listed as follows:

- Basic software structure and function as well as their relationship.

- Third-party library components and composition: shown in a dependency graph, each with a required minimum version number.
- Programming languages: what specific languages are used and their minimum language version.
- Compilers: what compilers and versions are required by the software package.
- 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 classify each of the hardware components into three categories: (i) Mandate, (ii) performance critical, or (iii) portable but slow.
- Communication layers: 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.

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)). This research used resources of the Oak Ridge Leadership Computing Facility, located in the National Center for Computational Sciences at Oak Ridge National Laboratory (ORNL), which is managed by UT-Battelle LLC for the Department of Energy under contract DE-AC05-00OR22725.

REFERENCES

Bader, D., Collins, W., Jacob, R., Jones, P., Rasch, P., Taylor, M., ... & Williams, D, 2014. Accelerated climate modeling for energy (ACME) project strategy and initial implementation plan.
Bastian, M., Sebastien H., and Mathieu J., 2009. Gephi: an open source software for exploring and manipulating networks. Icwsm. 8, 361-362.
Bush, W.R., Jonathan D.P., and David J.S., 2000. A static analyzer for finding dynamic programming errors. Software-Practice and Experience 30.7, 775-802.
Calcote, J., 2010. Autotools: A Practitioner's Guide to GNU Autoconf, Automake, and Libtool. No Starch Press.
Dor, N., Michael R., and Mooly S., 2003. CSSV: Towards a realistic tool for statically detecting all buffer overflows in C. ACM Sigplan Notices. Vol. 38. No. 5. ACM, 155-167
Gobeille, R., The fossology project. Proceedings of the 2008 international working conference on Mining software repositories. ACM., 47-50
Gough, B., 2009. GNU scientific library reference manual. Network Theory Ltd.
Kim, J., Esler, K., McMinis, J., Clark, B., Gergely, J., Chiesa, S., 2014. QMCPACK simulation suite.
Knüpfer, A., Brunst, H., Doleschal, J., Jurenz, M., Lieber, M., & Nagel, W. E, 2008. The vampir performance analysis tool-set. Tools for High Performance Computing. Springer. Berlin. Heidelberg., 139-155.
MacKenzie, D., Roland M., and Noah F., 1994. Autoconf: Generating automatic configuration scripts.
MacKenzie, D., Tom T., and Alexandre D., 1995. GNU Automake. User Manual, for Automake version 1.
Martin, K., and Bill H., 2010. Mastering CMake: a cross-platform build system. Kitware.
Maxwell, R.M., Kollet, S.J., Smith, S.G., Woodward, C.S., Falgout, R.D., Ferguson, I. M., ... & Ashby, S. 2009., ParFlow user's manual. International Ground Water Modeling Center Report GWMI1.2009., 129.
Ombredanne, P. et al., 2016. scancode-toolkit. GitHub repository, https://github.com/nexB/scancode-toolkit.
Schuberth, S. et al., 2017. oss-review-toolkit. Github repository, https://github.com/heremaps/oss-review-toolkit.
Seindal, R., 1997. GNU m4, version 1.4. Free Software Foundation 59.
Turner, J. et al., 2017. Exascale Simulation for Additive Manufacturing (ExaAM)., GitHub repository, https://github.com/ExascaleAM
Xie, Y., Chou, A., and Dawson E., 2003. Archer: using symbolic, path-sensitive analysis to detect memory access errors. ACM SIGSOFT Software Engineering Notes 28.5., 327-336.
Van Heesch, D., 2008. Doxygen: Source code documentation generator tool. http://www.doxygen.org.
Van Hoorn, A., Jan W., and Wilhelm H., 2012. Kieker: A framework for application performance monitoring and dynamic software analysis. Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering. ACM, 247-248.
Wilde, N., Ross H., and Scott H., 1989. Dependency analysis tools: reusable components for software maintenance. Software Maintenance, Proceedings.. IEEE, 126-131.
Wilhelm, M., and Stephan D., 2005. Dependency viewer-a tool for visualizing package design quality metrics. Visualizing Software for Understanding and Analysis. VISSOFT. IEEE., 1-2
Wu, Y., Roland HC.Y., and Rajiv R., 2010. Comprehending module dependencies and sharing. Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2. ACM, 89-98
Zirkelbach, C., Wilhelm H., and Leslie C., 2015. Combining Kieker with Gephi for Performance Analysis and Interactive Trace Visualization, 26-28.
Zitser, M., 2003. Securing software: An evaluation of static source code analyzers. Diss. Massachusetts Institute of Technology.