CAASCADE: A System for Static Analysis of HPC Software Application Portfolios

M. Graham Lopez[1(✉)], Oscar Hernandez[1], Reuben D. Budiardja[2], and Jack C. Wells[2]

[1] Computer Science and Mathematics Division, Oak Ridge National Laboratory, Oak Ridge, TN 37831, USA
{lopezmg,oscar}@ornl.gov
[2] National Center for Computational Sciences, Oak Ridge National Laboratory, Oak Ridge, TN 37831, USA
{reubendb,wellscj}@ornl.gov

Abstract. With the increasing complexity of upcoming HPC systems, so-called "co-design" efforts to develop the hardware and applications in concert for these systems also become more challenging. It is currently difficult to gather information about the usage of programming model features, libraries, and data structure considerations in a quantitative way across a variety of applications, and this information is needed to prioritize development efforts in systems software and hardware optimizations. In this paper we propose CAASCADE, a system that can harvest this information in an automatic way in production HPC environments, and we show some early results from a prototype of the system based on GNU compilers and a MySQL database.

1 Introduction

Heterogeneous architectures and complex system design have been consistent challenges for the high-performance computing (HPC) applications community. For example, in the ongoing CORAL project [4] and Exascale Computing Project (ECP) [10], HPC researchers, U.S. Department of Energy computing facilities,

A. Bhatele et al. (Eds.): ESPT/VPA 2017/2018, LNCS 11027, pp. 90–104, 2019.
https://doi.org/10.1007/978-3-030-17872-7_6

and system builders are engaged in designing the system software layers to tightly couple with both low-level hardware and high-level applications. In order to better inform such efforts, often referred to as "co-design," we have to answer specific questions about how applications are using current HPC architectures with detailed, quantitative data as evidence.

Currently in the HPC community, we have insufficient ways to know in quantitative detail which system software features are required by user applications; we most often rely on single-use, labor-intensive efforts [33], "institutional knowledge", or written survey responses and anecdotal input from developers [15]. This knowledge is tethered to the developers who have intimate knowledge of the codes, and current tools are used on a subset of applications providing either very narrow, application-specific views of the source code and performance traits that are not well-suited for inter-application reasoning or broad summaries that lose the detail needed for research and system design. This absence of quantitative application information at HPC centers leads to intuition-based engineering and is increasingly identified as an HPC community challenge with calls for structured responses emerging within community forums [31].

There are simple questions about the distribution of HPC applications that we cannot answer quickly and accurately in production application environments, like which programming language features, parallelization methods, libraries, and communication APIs are used commonly across HPC applications. These questions become even more urgent for the documentation of application requirements for next generation HPC systems, the planning of long-term computer science research programs to fill capability gaps, and in the execution of scientific applications readiness programs that prepare codes to accomplish large-scale science on upcoming systems. Program characteristics such as data structures layouts, data access patterns, type of parallelism used, profitable compiler optimizations and runtime information, need to be captured in a systematic and transparent way to the user, so that conclusions can be made at an HPC center-wide level.

To provide this currently unavailable information, we propose a method to automate the collection of application program characteristics from compiler-based tools and enable knowledge discovery and feature detection from this data. Since compilers know everything that is necessary about a source code to lower it to a resulting executable on a given architecture, we are working to create a curated database to provide convenient access to information harvested directly from compiler intermediate representations to enable data analytics techniques on application source code to inform ongoing HPC research and co-design activities. These cross-application analyses will lead to a quantitative understanding of the overall HPC application landscape and where high-value opportunities lie for development of system software and tools.

Having the ability to answer the questions similar to those above will enable evidence-based support for the HPC research community, standards committees, and system vendors. Exploring these issues in depth will allow researchers to gain continued and deeper insights into these issues as we co-design applications with

upcoming exascale software and hardware architectures as part of ECP. Our community will be better equipped to develop tools, inform hardware development and standards committees, and understand HPC science application needs. Designing HPC systems and vendor engagement will be facilitated by a complete and detailed understanding of real application practices and requirements across the breadth of HPC applications.

In order for this knowledge to be useful in the near and longer-term future, it needs to be both accurately harvested with reliable tools, curated for data quality, and made easily accessible to a variety of users. Rather than relying on one-off research and limited-use implementations, our approach uses as a reference implementation industry-standard tools that are already widely used on HPC platforms and can handle the complexities of all full HPC applications rather than miniapps or only specific applications, while being totally transparent from the user. The GNU Compiler Collection (GCC) is today able to compile the in-production version of most current HPC codes, and it supports both the OpenMP and OpenACC programming models. As the Clang and Flang LLVM front-ends continue to mature on upcoming systems and become adopted by production HPC application users, we envision porting our analysis tools to that toolchain as well. Finally, we have worked with PGI to implement this data extraction in their compiler suite, with the `-Msummary` flag made available in the PGI 17.7 release [7]. We hope to engage with other HPC compiler vendors in the near future and come up with defacto specifications for parallel program static analysis information.

2 Background and Related Work

Various tools have been developed to capture program information, but they are not commonly used for application data collection on production systems because they are either not fully automated (e.g. transparent to the user), have high barriers to entry for users, not able to handle full production application codebases, require significant user intervention (e.g. code restructuring, working with tools experts), and/or they are not available on all platforms.

OpenAnalysis [43] was an attempt to create a database of program analysis that can be reused across compilers or tools. It relied on Open64 [24] and ROSE [36] compiler components, but neither of these are widely used by production applications across HPC centers. The TAU Program Database Toolkit [37] captures program structure and stores it in the PDB format which is used for instrumenting the source code. However, this requires adding extra steps in the build system and parsing the application with PBT front-ends that may require program refactoring. The HPCToolkit [12] hpcstruct component gathers some program traits from the binaries of applications by trying to reconstruct specific constructs like loop nests, however it cannot detect the higher level features of languages due to information loss during lowering.

There have been compiler-based tools with advanced analysis capabilities. Tools such as ROSE [46], Hercules [35], TSF [21] and RTalk [25] store program

analysis information with the goal of applying transformation-based recipes that contain static or run-time information of the code. CHiLL, together with Active Harmony [45] focus on parameter selection and compiler heuristics for auto-tuning. The Klonos [26] tool extracts sequences of operators from the intermediate representation of compilers to find similarity between the codes, but the resulting information is difficult to relate back to the source code beyond the procedure functionality. These tools either do not cover the full spectrum of HPC languages or are maintained as research tools not intended to be used in production, and their goal is not to be totally transparent from the user, as they are meant to interact with the user.

The Collective Tuning project [30] aims to create a database of program structure features and find compiler optimizations for performance, power, and code size. The main goal of the now deprecated [8] GCC plugin-based MILEPOST project from cTuning was to collect program features for the purpose of feeding these back to the compiler optimizer, instead of being made understandable for human researcher consumption. However, it was the efforts of cTuning's Interactive Compilation Interface [6] project that contributed to GCC's plugin infrastructure that we now use for CAASCADE.

Dehydra [1] and Treehydra [2] are analysis plugins that expose different GCC intermediate representations intended for simple analyses and "semantic grep" applications. Unfortunately, they have only limited Fortran90 support, and the output hides important application information. Pliny [27] is a project that focuses on detecting and fixing errors in programs, as well as synthesizing reliable code from high-level specifications. It relies on mining information and statistical information and is still in the early research stage and currently doesn't support Fortran. Finally, tools such as XALT/ALTD [13,22], PerfTrack [34], Oxbow/-PADS [41], IPM [29], and HPC system scheduling information provide system environment, linkage information (e.g. for library detection) runtime and performance information that is complementary to application source code features. As discussed below, we intend for CAASCADE to interface closely with these related sources of application information.

3 Design and Methods

The core of CAASCADE consists of: the compiler-based static analysis and the data storage and analytics backend. Information about various aspects of the application source code is harvested directly from the compiler's intermediate representations (IR) and converted into a format that can be ingested into a SQL database or analytic engines such as Apache Spark [47]. CAASCADE produces program information for each compilation unit and inserts the information in the object file. At link time, it then retrieves the program information from every object file and stores this information in the database. A symbol identifier is inserted in the application executable binary to identify it with the program information stored in the database.

3.1 GNU Compiler Plugin Implementation

To prototype the static application analysis, we used the built-in plugin infrastructure [5] from the GNU Compiler Collection (GCC) to extract information directly from the compiler's IR and data structures. There are about thirty plugin callback hooks to trigger plugin execution, that span locations from just before a new translation unit pass is started to the majority of these locations being among the various lowering and optimization passes.

For gfortran, GNU doesn't currently provide any plugin callback in the Fortran front-end. We found it easiest to add our own call-back to the gfortran front-end that triggers right after the processing of a translation unit is completed (after parsing but before any lowering). For the C/C++ front-end as well, we added our own hook into the g++ front-end right after a translation unit has been finalized, and piggy-back onto the translation-unit tree dumper (enabled with the `-fdump-translation-unit` flag to g++), where we reuse the internal tree traversal engine, but insert our own data extraction and processing routines.

By placing our plugin execution carefully within the GCC front-ends, we know that all of the parsing and abstract syntax tree (AST) building has been completed, but lowering has not yet taken place. This approach will also help us store the compiler's internal data structures to communicate across multiple levels of intermediate representation and relate the analysis back to the source code. Currently, we primarily target the AST level IR as it is most directly relatable to the original user-written source code. However, our goal is to enable the extraction of program information and analysis from multiple levels of the intermediate representation while mapping them to the source code.

High-Level Languages. Each translation unit is characterized by the invocation of a GCC language front-end, so to keep track of the proportion of each high-level language being used in an application, we can accumulate statistics about executable statements and data declarations within each translation unit. By working at the AST level, it is possible to eliminate inconsistencies such as comments, whitespace, bracket placement, or line-continuations. Additionally, based on the features that we see in executable statements or declarations and classifying them according to the language standard needed to support those features, we can determine the proportion of each language standard being used and the coverage across those standards. Table 1 shows the information the compiler is collecting about the source code for Fortran applications.

In addition to the metrics above, the plugins collect information regarding variable and data structure information from the application; a list of these metrics is shown in Table 2.

Parallelization Methods. Our plugins understand directives from both OpenMP [39] and OpenACC [42], as well as Message Passing Interface (MPI) [38] library calls. This allows us to easily detect when we are within a parallel directive's lexical extent, and which variables and types are being used in inter-node

communication via MPI or directive `data` clauses for transfers between host and discrete accelerator memories. As with the high-level languages, we can also detect the proportion and coverage of each standard being used, based here on matching the directives or calls that are present in the code with those in the specification versions. For understanding directives, we depend on the compiler's native support and so are constrained to versions of OpenMP and OpenACC that have been implemented in GCC. To handle MPI, we treat it as any other library (as discussed below). Table 2 shows the program information we collect about the directives parallelization method.

Table 1. Translation unit and procedure information

Translation unit	Procedures
compiler version	subroutine name
programming language/model	# of exec statements
module/class/typedef	# of loops
main program name	max loop nest depth
line numbers	# call statements
	list of call chains
	# use modules
	total module variables
	list of module variables
	list of module subroutines
	# of symbols
	# symbols in other namespaces
	# of namelists
	# of statements
	classification by statement types
	modules used by subroutine
	classification of statements per standard

Libraries. While there is a practically unlimited number of libraries that could be used in a code, we are interested in common HPC libraries that have high reuse across applications, are critical for an application's performance and portability across HPC architectures, and typically require a large effort (both by the community and hardware vendors) to optimize for various platforms. As a starting point, we chose a sample of numerical (BLAS [14], LAPACK [16], FFTW [28], PETSc [18–20]), communication (MPI [38], SHMEM [23]), and data management (HDF5 [44], ADIOS [11], NetCDF [40]) libraries.

For C/C++, this task is made easier by the necessity of include files which become part of the analyzed translation unit. By descending into recognized include files, symbol names can be gathered that are known to be part of the

library, and their usage examined along with the application's declared data representations, for example as type information passed as part of function parameters. However, in the case of Fortran, it is necessary to separately gather, compile, and store information about the libraries of interest, and then compare the information from application compilation to that previously seen during the separate library compilation. For this reason, the source code metadata pertaining to the libraries listed above must be stored along with the data from applications of interest.

Table 2. Data structure and parallelization method information

Variable/Data structure information	Parallelization method information
# variables	# OpenMP directives
# array variables	# statements inside OpenMP
# co-array variables	# OpenMP threadprivate variables
# pointer variables	# OpenMP UDR variables
# contiguous variables	# OpenMP declare target variables
# target variables	# OpenACC directives
# allocatable variables	# statements inside OpenACC
# artificial variables	# OpenACC subroutine
# asynchronous variables	# OpenACC declare create variables
# optional variables	# OpenACC declare copyin variables
# dummy variables	# OpenACC declare deviceptr variables
# protected variables	# OpenACC declare device_resident variables
# volatile variables	# OpenACC declare link variables
# abstract variables	
# implicit type variables	
# in namelist variables	
# external variables	
# parameters	
# common block variables	
# derived types	
# derived types with components	
# derived types with direct components	
# derived types with indirect components	
# derived types with array components	
# derived types with allocatable components	
# derived types with pointer components	
# derived types recursive	

3.2 Database Infrastructure

As compilation proceeds and the plugin gathers data from the IR, this data is stored in an application-independent way so that it is available for queries about the application and its design. The goal is to make the data more accessible than the source code or the raw compiler IR, but flexible enough that it minimizes presuppositions about particular questions that might be asked about an application's implementation.

Application Data. In order to provide the most general representation of the data while making it structured enough that non-compiler experts can explore and use it, we use an SQL database with a normalized schema. The schema holds administrative data about the build platforms and application versions, as well as the source code metadata itself. This enables comparison and differentiation for the data collected depending on the target platforms of the application compilation, e.g. if specialized features are guarded with `#ifdef`'s. Our current version stores a linkage and compile table to store program information. The compiler output from both GNU and PGI is stored in JSON format. The next version of the tool will refine the database schema to the compilation information from both compilers into a single schema.

To ensure support in different HPC environments, we allow several transmission methods to store to the database the results gathered by our plugins. We leverage the XALT [13,22] transmission machinery to easily accomplish this. The most direct way is by making an SQL connection from the compiler plugin itself and inserting the streaming results into the database. The plugins can also create an intermediate JSON file which can be parsed at some later time for consumption by the database. Finally, the plugin can elect to send results via syslog to a logging server which can then be parsed for database storage. The latter two methods are useful if direct database connection is not possible or is undesirable, for example due to security considerations that require the database server to be in a different network enclave, or due to performance considerations by avoiding the higher latency often associated with direct database queries on high-load machines.

Front-End Access. Storing this data in the database also enables advanced queries and post-processing analysis to be performed at a later time in order to gain more insight about the applications. For example, one can get historical perspectives about how data structures and directives in the source code have evolved by looking at the plugin compilation data over time. One can also do queries across different databases about the application (e.g. from a user accounts/project database, a job submission database, an XALT database [13,22], etc.). An inter-source-code analysis across all files for an application can be done to gain an overall understanding of all the data structures and directives used by an application.

Some users may not want to interact directly with SQL queries, so we are building a front-end website that allows the user to get basic insights about the application data. We provide graphical representations for some of the major statistics that are gathered and stored by the system. An example showing language usage, parallelization strategies, and data structure compositions is shown in Fig. 1 for the ACME [17] application.

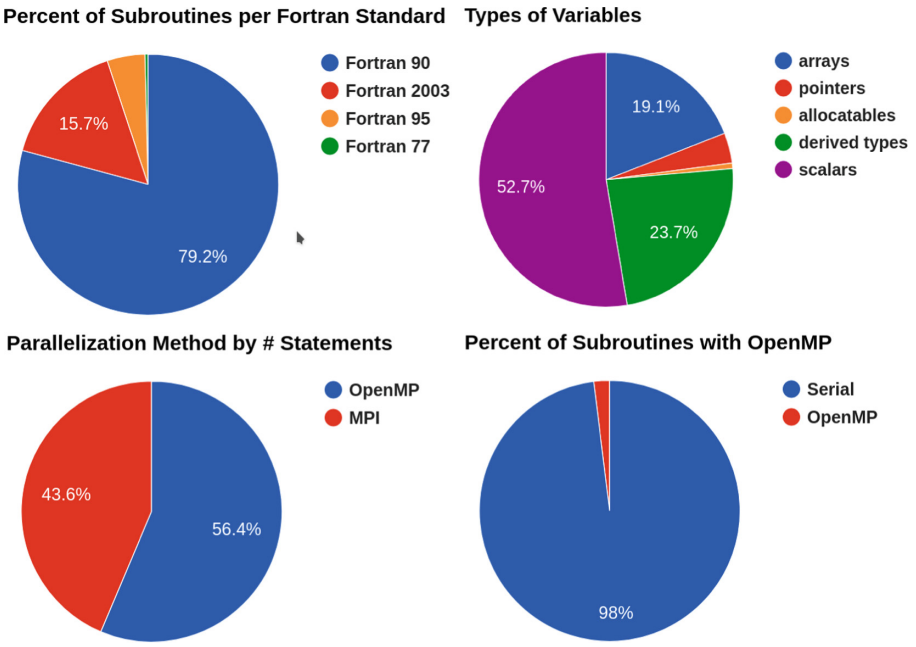

Fig. 1. High-level information such as Fortran language standard (top left), the type of variables (top right), OpenMP and MPI parallelization methods by the number of statements (bottom left), and subroutines with OpenMP pragmas (bottom right) in ACME as collected by our tools.

4 Results

As a prototype of these ideas, we have implemented the compiler-based static analysis in the GCC gfortran and g++ frontends, and through collaboration with PGI in the pgfortran and pgcc frontends as of release 17.7. We have designed an SQL schema as described in Sect. 3.2, which we use in a database alongside the XALT system installed at the Oak Ridge Leadership Computing Facility (OLCF). In the following sections, we demonstrate a sample of some of the basic statistics that can be gathered using these tools on full in-production HPC applications.

We take the Accelerated Climate Modeling for Energy (ACME) [17] as one of the first applications on which we exercised our tools. To make our tools work transparently on the system, we created a simple wrapper for the GCC compiler executables to automatically enable our custom plugin. This avoids having to modify the application build system with specific flags. Our tools also automatically insert the collected compile-time information back into the object files. During linking, a linker wrapper goes through these object files to gather these data using the selected transmission method (see Sect. 3.2).

As a sample of the information that is automatically pre-generated by our front-end website, Fig. 1 shows high-level information from the ACME application such as the Fortran language standard usage in the code, the distribution of the type of variables and parallelization methods (OpenMP and/or MPI).

In Fig. 1, the top left panel shows the relative usage of various Fortran standards, which (combined with specifics on the features used) gives an indication of the support required by compilers for HPC architectures. The lower left panel gives a high-level description of the overall "MPI+X" parallelization scheme being used in this application. While MPI and OpenMP express their respective forms of parallelism differently, this comparison might give an overview of the relative effort being expended on each type of parallelism – in this case, it is essentially equally distributed. The top right and bottom right panels indicate

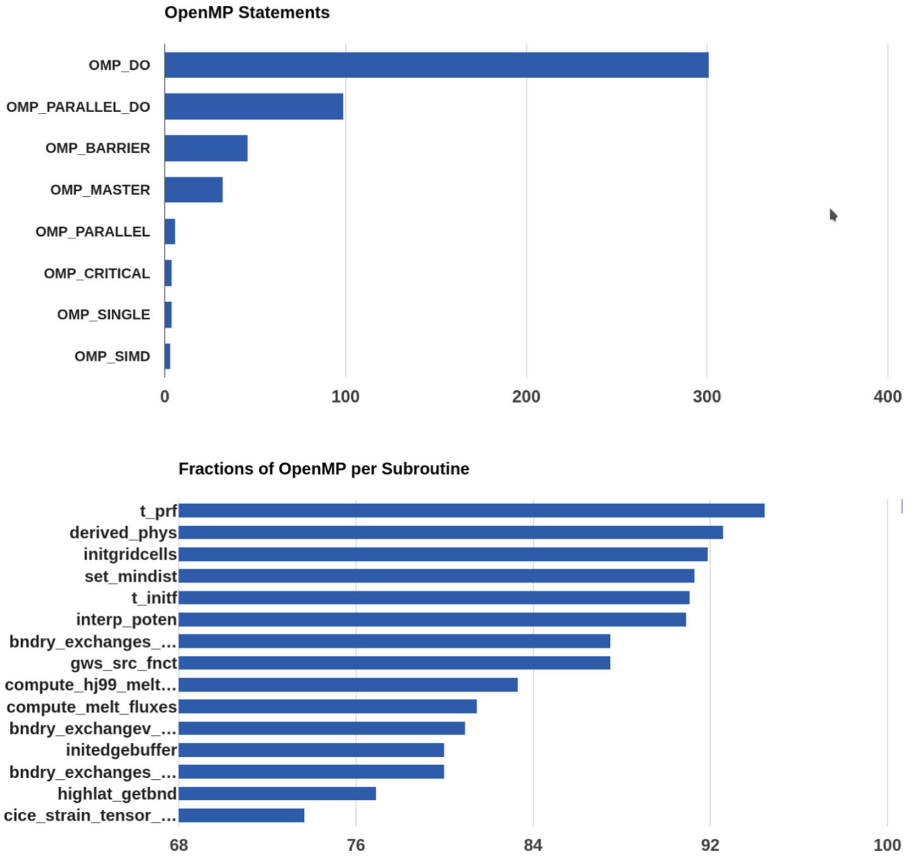

Fig. 2. Summary information about OpenMP usage in the ACME application. The usage of specific OpenMP statements is shown in the top panel, and the proportion of code covered within OpenMP lexical extents is shown per subroutine in the bottom panel. On the bottom panel, only the top fifteen subroutines are shown for clarity.

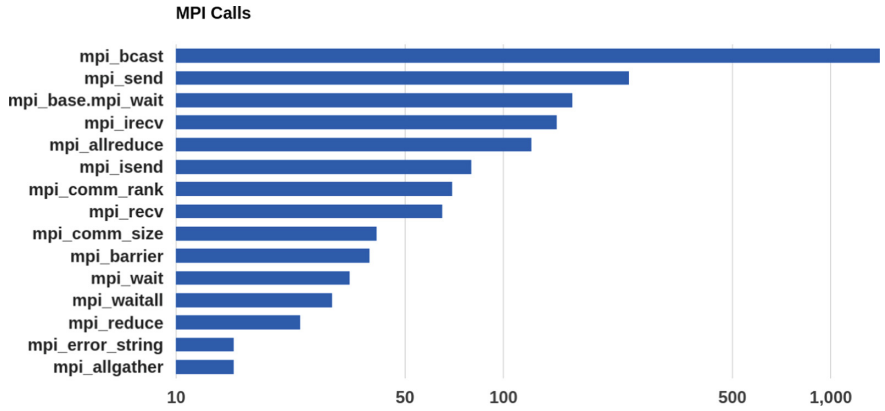

Fig. 3. MPI routines used in the ACME application. The frequency of calls for each MPI routine is shown. Only the fifteen most called routines are shown for clarity.

porting efforts that might be required for new parallelization schemes or architectures by showing the usage of various types of data structures being used, and the percentage of subroutines that most probably need to be considered.

Figure 2 goes one step deeper into the usage of OpenMP in ACME, with the top panel showing the coverage of OpenMP features being used, and their frequency in the source code. The bottom panel indicates the concentration of intranode parallelism by showing the percentage of code covered by OpenMP lexical extent per subroutine.

Similar information regarding MPI calls can also be easily obtained. Figure 3 shows the frequency of MPI routines used in the application. It is relatively trivial to expand this information gathering across not only applications, but also different libraries. Although in this case we only show example for one application (ACME) and one library (MPI), it is easy to imagine that having this information in aggregate across multiple applications and libraries will give insights into co-design efforts.

5 Conclusions and Future Work

In this paper, we have outlined a strategy to fill a current gap in the HPC application development and co-design ecosystems. Compilers know everything necessary about a source code to determine the behavior of the executable on a given architecture. We have shown that it is possible to extract this information from compilers and make it accessible and useful for inter-application analyses. In the near term, we have received requests for this data for use in designing and extending parallel APIs in both programming models and well-known HPC libraries. We also hope that this information will be useful in the upcoming exascale platform designs by shedding light on how application developers have

been using the current leadership architectures, as well as prototyping their algorithms on new hardware.

In order to increase the deployment flexibility of the system and decouple the data analysis and storage phases, we envision using the DWARF [3] binary data format, which is an extensible open-standard format for storing information in binaries generated by most modern compilers both proprietary and open-source. DWARF can be used to store the static analysis information together with the generated application binary, which could then be extracted by existing, standard-conforming binary manipulation utilities. This would allow for a portable and standardized way for other compiler implementers (including proprietary and closed-source) to participate in the system.

For maximum coverage, the prototype system as described in this paper is being installed on the Oak Ridge Leadership Computing Facility (OLCF) Titan [32] and early-access Summit [9] systems to automatically gather and store application data at compile time from participating users on an opt-in/out basis. It is necessary to work with early adopters to determine the appropriate level of data anonymization and sanitation before making the data publicly available outside of OLCF. Systems usage and user job information is already being stored in systems like XALT, and we have coupled our SQL schema to be easily queried together with the dynamic linkage and job submission information already being captured by XALT.

Furthermore, coupling detailed yet generalizable information from static analysis and runtime performance analysis will greatly increase the efficacy of both, especially for inter-application statistics. For example, combining hotspot analysis with data structure layout information should lead to optimization opportunities in compilers, runtimes, and programming model design.

Structuring the data in an application- and compiler-agnostic way and storing it in an SQL database that can accommodate flexible queries is essential to servicing previously intractable questions about application source code and programming model usage. However, even more complex data analytics is made possible by using purpose-built frameworks like Apache Spark. Through ongoing collaborations, we are investigating these techniques to answer "fuzzier" questions about topics such as automatic parallel computational motif usage and application evolution over time and hardware architectures.

It is hoped that tools such as the one presented here, when coupled with advanced data analytics techniques, will lay the foundation for future research in sophisticated methods for adapting high-performance applications to various types of architectures, such as using machine learning techniques for porting applications. Additionally, this work will provide better insight into how HPC applications are evolving over time and across disparate architectures, e.g. through quantitative metrics that can be captured as applications transition to exascale platforms.

Acknowledgements. This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, under contract number DE-AC05-00OR22725.

This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

Research sponsored by the Laboratory Directed Research and Development Program of Oak Ridge National Laboratory, managed by UT-Battelle, LLC, for the U. S. Department of Energy. The project was sponsored via the LDRD project 8277: "Understanding HPC Applications for Evidence-based Co-design".

References

1. https://developer.mozilla.org/en-US/docs/Archive/Mozilla/Dehydra
2. https://developer.mozilla.org/en-US/docs/Archive/Mozilla/Treehydra
3. http://dwarfstd.org/Dwarf5Std.php
4. CORAL fact sheet. http://www.anl.gov/sites/anl.gov/files/CORAL
5. GNU Compiler Collection (GCC) Internals: Plugins. https://gcc.gnu.org/onlinedocs/gccint/Plugins.html
6. Interactive compilation interface. http://ctuning.org/wiki/index.php/CTools:ICI
7. PGI Compiler & Tools. http://www.pgroup.com/index.htm
8. Reproducing MILEPOST project. https://github.com/ctuning/ck/wiki/Reproducing-MILEPOST-project
9. Summit: scale new heights. Discover new solutions. https://www.olcf.ornl.gov/summit/
10. The Exascale Computing Project. https://exascaleproject.org
11. Abbasi, H., Lofstead, J., Zheng, F., Schwan, K., Wolf, M., Klasky, S.: Extending I/O through high performance data services. In: 2009 IEEE International Conference on Cluster Computing and Workshops, pp. 1–10, August 2009. https://doi.org/10.1109/CLUSTR.2009.5289167
12. Adhianto, L., et al.: HPCTOOLKIT: tools for performance analysis of optimized parallel programs. Concurr. Comput.: Pract. Exp. **22**(6), 685–701 (2010). https://doi.org/10.1002/cpe.1553
13. Agrawal, K., Fahey, M.R., McLay, R., James, D.: User environment tracking and problem detection with XALT. In: Proceedings of the First International Workshop on HPC User Support Tools, HUST 2014, pp. 32–40. IEEE Press, Piscataway (2014). https://doi.org/10.1109/HUST.2014.6
14. Blackford, L.S., et al.: An updated set of basic linear algebra subprograms (BLAS). ACM Trans. Math. Softw. **28**(2), 135–151 (2002). https://doi.org/10.1145/567806.567807
15. Anantharaj, V., Foertter, F., Joubert, W., Wells, J.: Approaching exascale: application requirements for OLCF leadership computing. Oak Ridge Leadership Computing Facility Technical report ORNL/TM-2013/186 (2013)
16. Anderson, E., et al.: LAPACK Users' Guide, 3rd edn. Society for Industrial and Applied Mathematics, Philadelphia (1999)
17. Bader, D.B., et al.: ACME Pre-Release Documentation (2017). http://climatemodeling.science.energy.gov/acme/information-for-collaborators
18. Balay, S., et al.: PETSc Web page (2016). http://www.mcs.anl.gov/petsc
19. Balay, S., et al.: PETSc users manual. Technical report ANL-95/11 - Revision 3.7. Argonne National Laboratory (2016). http://www.mcs.anl.gov/petsc

20. Balay, S., Gropp, W.D., McInnes, L.C., Smith, B.F.: Efficient management of parallelism in object oriented numerical software libraries. In: Arge, E., Bruaset, A.M., Langtangen, H.P. (eds.) Modern Software Tools in Scientific Computing, pp. 163–202. Birkhäuser Press, Boston (1997)
21. Bodin, F., Mével, Y., Quiniou, R.: A user level program transformation tool. In: Proceedings of the 12th International Conference on Supercomputing, pp. 180–187. ACM (1998)
22. Budiardja, R., Fahey, M., McLay, R., Don, P.M., Hadri, B., James, D.: Community use of XALT in its first year in production. In: Proceedings of the Second International Workshop on HPC User Support Tools, HUST 2015, pp. 4:1–4:10. ACM, New York (2015). https://doi.org/10.1145/2834996.2835000
23. Chapman, B., et al.: Introducing OpenSHMEM: SHMEM for the PGAS community. In: Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model, PGAS 2010, pp. 2:1–2:3. ACM, New York (2010). https://doi.org/10.1145/2020373.2020375
24. Chapman, B., Eachempati, D., Hernandez, O.: Experiences developing the OpenUH compiler and runtime infrastructure. Int. J. Parallel Program. **41**(6), 825–854 (2013). https://doi.org/10.1007/s10766-012-0230-9
25. von Dincklage, D., Diwan, A.: Integrating program analyses with programmer productivity tools. Softw. Pract. Exp. **41**(7), 817–840 (2011). https://doi.org/10.1002/spe.1035
26. Ding, W., Hsu, C.H., Hernandez, O., Chapman, B., Graham, R.: KLONOS: similarity-based planning tool support for porting scientific applications. Concurr. Comput. Pract. Exp. **25**(8), 1072–1088 (2013). https://doi.org/10.1002/cpe.2903
27. Feser, J.K., Chaudhuri, S., Dillig, I.: Synthesizing data structure transformations from input-output examples. SIGPLAN Not. **50**(6), 229–239 (2015). https://doi.org/10.1145/2813885.2737977
28. Frigo, M., Johnson, S.G.: The design and implementation of FFTW3. Proc. IEEE **93**(2), 216–231 (2005). https://doi.org/10.1109/JPROC.2004.840301
29. Fuerlinger, K., Wright, N.J., Skinner, D.: Effective performance measurement at petascale using IPM. In: 2010 IEEE 16th International Conference on Parallel and Distributed Systems, pp. 373–380, December 2010. https://doi.org/10.1109/ICPADS.2010.16
30. Fursin, G., Lokhmotov, A., Plowman, E.: Collective knowledge: towards R&D sustainability. In: Proceedings of the 2016 Conference on Design, Automation & Test in Europe, DATE 2016, pp. 864–869. EDA Consortium, San Jose (2016). http://dl.acm.org/citation.cfm?id=2971808.2972009
31. Gibbs, T.: Accelerate the path to exascale: best practices to acquire and analyze HPC application workload data (2016). http://www.isc-hpc.com/isc16_ap/sessiondetails.htm?t=session&o=328&a=select&ra=personendetails
32. Joubert, W., et al.: Accelerated application development: the ORNL Titan experience. Comput. Electr. Eng. **46**, 123–138 (2015). https://doi.org/10.1016/j.compeleceng.2015.04.008
33. Joubert, W., Su, S.Q.: An analysis of computational workloads for the ORNL Jaguar system. In: Proceedings of the 26th ACM International Conference on Supercomputing, ICS 2012, pp. 247–256. ACM, New York (2012). https://doi.org/10.1145/2304576.2304611

34. Karavanic, K.L., et al.: Integrating database technology with comparison-based parallel performance diagnosis: the PerfTrack performance experiment management tool. In: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing, SC 2005, p. 39. IEEE Computer Society, Washington, DC (2005). https://doi.org/10.1109/SC.2005.36
35. Kartsaklis, C., Hernandez, O., Hsu, C.H., Ilsche, T., Joubert, W., Graham, R.L.: HERCULES: a pattern driven code transformation system. In: 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum, IPDPSW, pp. 574–583. IEEE (2012)
36. Liao, C., Lin, P.H., Quinlan, D.J., Zhao, Y., Shen, X.: Enhancing domain specific language implementations through ontology. In: Proceedings of the 5th International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing, WOLFHPC 2015, pp. 3:1–3:9. ACM, New York (2015). https://doi.org/10.1145/2830018.2830022
37. Lindlan, K.A., et al.: A tool framework for static and dynamic analysis of object-oriented software with templates. In: ACM/IEEE 2000 Conference on Supercomputing, p. 49, November 2000. https://doi.org/10.1109/SC.2000.10052
38. MPI Forum: MPI: a message-passing interface standard. Technical report. MPI Forum, Knoxville, TN, USA (1994)
39. OpenMP Architecture Review Board: OpenMP application program interface version 4.5, November 2015. http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf
40. Rew, R.K., Davis, G.P., Emmerson, S.: NetCDF user's guide, an interface for data access, version 2.3, April 1993
41. Sreepathi, S., et al.: Application characterization using Oxbow toolkit and PADS infrastructure. In: 2014 Hardware-Software Co-Design for High Performance Computing, pp. 55–63, November 2014. https://doi.org/10.1109/Co-HPC.2014.11
42. OpenACC Standard: OpenACC - directives for accelerators. http://www.openacc-standard.org
43. Strout, M.M., Mellor-Crummey, J., Hovland, P.: Representation-independent program analysis. SIGSOFT Softw. Eng. Notes **31**(1), 67–74 (2005). https://doi.org/10.1145/1108768.1108810
44. The HDF Group: Hierarchical data format version 5 (2000–2017). http://www.hdfgroup.org/HDF5
45. Tiwari, A., Chen, C., Chame, J., Hall, M., Hollingsworth, J.K.: A scalable auto-tuning framework for compiler optimization. In: IEEE International Symposium on Parallel & Distributed Processing, IPDPS 2009, pp. 1–12. IEEE (2009)
46. Willcock, J.J., Lumsdaine, A., Quinlan, D.J.: Reusable, generic program analyses and transformations. In: Proceedings of the Eighth International Conference on Generative Programming and Component Engineering, GPCE 2009, pp. 5–14. ACM, New York (2009). https://doi.org/10.1145/1621607.1621611
47. Zaharia, M., et al.: Apache spark: a unified engine for big data processing. Commun. ACM **59**(11), 56–65 (2016). https://doi.org/10.1145/2934664