CCA/EBT: Code Comprehension Assistance Tool for Evidence-Based Performance Tuning

Masatomo Hashimoto*
Software Technology and Artificial Intelligence Research Laboratory
Chiba Institute of Technology
Narashino, Chiba, Japan
m.hashimoto@stair.center

Masaaki Terai
Operations and Computer Technologies Division
RIKEN Advanced Institute for Computational Science
Kobe, Hyogo, Japan
teraim@riken.jp

Toshiyuki Maeda
Software Technology and Artificial Intelligence Research Laboratory
Chiba Institute of Technology
Narashino, Chiba, Japan
tosh@stair.center

Kazuo Minami
Operations and Computer Technologies Division
RIKEN Advanced Institute for Computational Science
Kobe, Hyogo, Japan
minami_kaz@riken.jp

ABSTRACT

Application performance tuning is still quite an art, despite advances in auto-tuning systems. Evidence-based performance tuning (EBT) aims at helping performance engineers gain and share evidence of performance improvement to make better decisions. As a step toward the goal, we have developed a tool, CCA/EBT, which assists performance engineers in comprehending source code written in Fortran, especially to identify loop kernels. The tool analyzes syntactic/semantic structures of source code and then provides outline views of the nested loops and the call trees, decorated with source code metrics. With the tool, 175 963 loops from a thousand computation-intensive applications have been explored. Based on the manual classification results of a randomly sampled subset of the loops, we have constructed an additional module for predicting loop kernels, which achieved cross-validated classification accuracy of 81%.

ACM Reference Format:
Masatomo Hashimoto, Masaaki Terai, Toshiyuki Maeda, and Kazuo Minami. 2018. CCA/EBT: Code Comprehension Assistance Tool for Evidence-Based Performance Tuning. In *Proceedings of High Performance Computing in Asia-Pacific Region (HPC Asia 2018).* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

To improve the performance of a scientific application, we have to identify its *computational kernels*, each of which is typically composed of one or more loops. Then various empirical attempts are made to achieve a high percentage of the theoretical peak performance of a given computing system. In general, application performance tuning is still a demanding task relying on experience and intuition, although a number of studies on auto-tuning systems are conducted for specific computational kernels such as stencil code, linear algebra solvers, and matrix multiplications [1], or even full applications [9]. Evidence-based performance tuning (EBT) [5] aims at helping engineers gain and share evidence of performance improvement to make better decisions. More specifically, our long-term goal is to construct a database of facts, or *factbase*, extracted from performance tuning histories of computational kernels such that we can search the database for promising optimization patterns that fit a given computational kernel.

In general, however, it is difficult to obtain detailed histories of performance tuning of computational kernels, since they would be thrown away as soon as the processes are finished. One approach is to extract such histories from repository hosting services such as GitHub[1]. As a step toward the goal, we have developed a tool, CCA/EBT, which assists performance engineers in comprehending source code written in Fortran, especially to classify loops into kernels and non-kernels.

The rest of the paper is organized as follows. Section 2 explains loop classification for identifying kernels. Then an overview of the tool is given in Section 3. After related work is reviewed in Section 4, Section 5 concludes the work.

2 CLASSIFICATION OF LOOPS

To improve the performance of a scientific application, comutational kernel, or kernel for short, are extracted from it first. Kernels can be further classified into several classes, each of which is related to expected efficiency, or a percentage of the theoretical peak performance (floating-point operations per second), and a typical tuning strategy [6]. We focus on the identification of a kernel composed of one or several nested loops, or a *loop kernel*. Kernels are identified based on static features extracted from the loops in the source code and/or based on dynamic features extracted from the

*Also with RIKEN Advanced Institute for Computational Science.

HPC Asia 2018, January 2018, Tokyo Japan

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

[1] https://github.com/

Table 1: Syntactic features of a loop

Abbrev.	Feature
FOp	Number of floating-point operations
St	Number of statements
Br	Number of branches
AR	Number of array references
DAR	Number of direct array references
IAR	Number of indirect array references
B/F	Bytes per flop
MLL	Maximum loop nest level

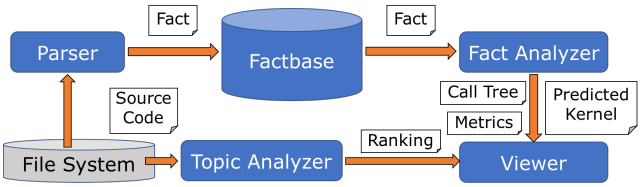

Figure 1: Overview of CCA/EBT

runtime performance data. Table 1 shows an example of a set of loop features we consult to determine whether a loop is a kernel or not.

A computation that a piece of kernel code describes can be characterized primarily by the amount of bytes of memory (RAM) accesses relative to floating-point operations, *B/F* for short, essentially required by the computation regardless of individual implementations.[2] Actual B/F of a specific code region is usually measured dynamically by profiling or tracing. While syntactically counting floating-point operations is relatively straightforward, estimating the volume of memory traffic is far from easy, as it would require cache behavior prediction. Instead of employing complex cache models, we employ the following estimation schemes [6].

ES0 assumes that data is shared in cache only among syntactically identical array references.

ES1 assumes that the data referenced by the array references that differ only by the first dimension are located in the same cache block.

ES2 assumes that the data referenced by the array references that differ only by the first dimension and/or additions or subtractions of constants at the second dimension are located in the same cache block.

For example, we assume by ES2 that `a(i1,j,k)` and `a(i2,j+1,k)` are located in the same cache block.

3 OVERVIEW OF CCA/EBT

Figure 1 gives an overview of the tool.

3.1 Fortran Parser

Extracting features of loops from source code requires its *abstract syntax tree (AST)* obtained by parsing. We employ a dedicated Fortran parser developed by Hashimoto and others [5]. The parser is based on language standards such as FORTRAN77, Fortran90, Fortran95, Fortran2003, and Fortran2008. It is also capable of handling the following.

Dialects The parser is capable of parsing dialects and language extensions made locally by compiler vendors such as IBM, PGI, and Intel.

Directives It can directly parse directives/constructs of the C preprocessor, OpenMP[3], OpenACC[4], OCL (Fujitsu), XLF (IBM), and DIR/DEC (Intel).

Partial failure It provides keep-on-parsing mode by making use of Menhir[5], a LR(1) parser generator, with error recovery function enabled to build the core of the parser.

Incomplete program fragments It is capable of parsing incomplete program fragments such as sequences of statements, which are typically included in other source files.

By virtue of the unusual features explained above, we can parse application programs without hooking the build process of the applications, which means that we can parse source files in any order without taking care of the dependencies among them. Instead, some dependencies caused by INCLUDE lines, `#include` directives, and USE statements may hinder the parser from determining types of some syntactic entities. As a result, AST nodes such as array elements, substrings, function references, or structure constructors may be left ambiguous. We disambiguate them later by resolving dangling references with the help of a database that contains information extracted from the whole source code.

3.2 Factbase

As mentioned above in Section 3.1, our parser requires deferred resolution of dangling references and/or ambiguous symbols. Moreover, a loop classification task requires call trees to obtain maximum loop nest level (MLL) seen in Section 2. We make use of a factbase, that is, a database for storing facts extracted from multiple source files. Reference/symbol resolution and feature extraction are performed by querying the factbase.

As the factbase queries may contain AST patterns and/or call graph patterns, it should be natural to use tree/graph databases rather than conventional relational databases. We employ an RDF (Resource Description Framework)[6] store, Virtuoso[7]. Instead of rigid database schemas, RDF stores require more flexible vocabularies, or *ontologies*. An ontology defines hierarchies of concepts such as "a *do-stmt* is a statement" and allows us to describe database queries concisely.

3.3 Fact Analyzer

By querying a factbase, the fact analyzer resolves dangling references and ambiguous symbols, constructs call trees, and extracts loop metrics. Queries are written in SPARQL[8], which is a standard query language for RDF stores. Roughly speaking, SPARQL is an extension of SQL with graph patterns that contain *query variables*. For more details on the queries, we refer the reader to the paper [6].

[2] Instead of B/F, we can use multiplicative inverse of B/F, or *operational intensity*, used in the roofline model [10] as long as it is used consistently.

[3] http://openmp.org/
[4] http://openacc.org/
[5] http://cristal.inria.fr/~fpottier/menhir/
[6] http://www.w3.org/RDF/
[7] http://virtuoso.openlinksw.com/
[8] http://www.w3.org/TR/sparql11-query/

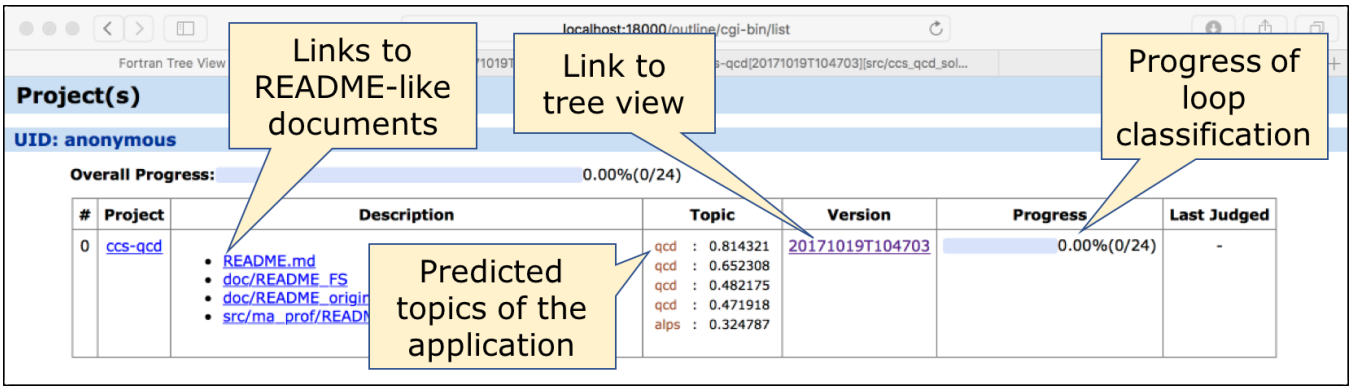

Figure 2: Project view

With the fact analyzer, Hashimoto and others analyzed 175 963 loops from a thousand computation-intensive applications hosted on GitHub [6]. From the set of loops, 100 were randomly sampled and then manually classified by experienced performance engineers with the aid of CCA/EBT. Based on the classification results, we have constructed an additional module for predicting loop kernels. We employed C-SVC (C-Support Vector Classification) in LIBSVM [3] from a data mining library called scikit-learn[9] to construct the predictive model by using the classification results as training data. The model achieved 20-fold cross-validated classification accuracy of 81% [6].

3.4 Topic Analyzer

To help performance engineers understand an application, we also analyze comments and variable names occurring in the source code for examining the topic or research field of the application. We constructed a topic model with *latent semantic indexing (LSI)* [4] based on 168 papers of scientific applications from several research fields such as quantum chemistry, astrophysics, and climate science.

3.5 Viewer

The viewer, implemented as a web application, provides three different views: project summary view, tree view and source code view. A snapshot of a project summary view is given in Figure 2, where automatically generated links to the documents whose names contain "README" are shown. The result of topic analysis is shown as a ranking of candidate applications, such as `qcd` and `alps`, similar to the project. You will see a link to the tree view as well as the progress of a user's loop classification performed in the tree view.

In a tree view, we can explore ASTs and call trees decorated with loop metrics. It is also possible to store users' loop classification results in a database behind the viewer. Figure 3 shows a snapshot of a tree view, where a predicted loop kernel decorated with extracted static features such as estimated B/F is highlighted over the outline of the AST and the call tree of an entire application. It should be noted that the size of a call tree may be infinite when it contains recursive calls. In the tree view, a procedure or function appears only at the deepest level of non-recursive calls. It should be noted also that we can select an estimation scheme for features AR, DAR, IAR, and B/F, where the suffixes 0, 1, and 2 of them indicate the estimation schemes ES0, ES1, and ES2, respectively.

Figure 4 gives a source code view that appears when a tree node is double-clicked, where array references are highlighted and quick look of the definitions appear on mouseover.

4 RELATED WORK

There are several commercial and open-source Fortran analysis tools: FORCHECK [2], Photran [7], and CamFort [8] to name a few. However, to the best of the authors' knowledge, there is no tool that is capable of predicting loop kernel nor of parsing a thousand of applications in a fully automated way.

5 CONCLUSION

As a step toward the long-term goal of helping performance engineers gain and share evidence of performance improvement, we have developed a tool, CCA/EBT[10], which assists performance engineers in comprehending source code written in Fortran, especially to classify loops into kernels and non-kernels. The tool analyzes syntactic/semantic structures of source code and then provides outline views of the nested loops and the call trees, decorated with source code metrics.

By using the tool, 175 963 loops from a thousand computation-intensive applications have been explored . Based on the manual classification results of a randomly sampled subset of the loops, we have constructed an additional module for predicting loop kernels. The kernel prediction module have not been extensively evaluated on the loops other than the sampled ones yet. They might have to be evaluated based on whether they can detect actual bottlenecks in an application, even though they were made relying only on manual classification results without measuring the runtime performance. Nevertheless, once a large number of kernels are predicted, we would be able to trace the change histories of them in an automated way [5], and hence to construct a database of performance tuning examples for the long-term goal.

[9] http://scikit-learn.org/

[10] https://github.com/ebt-hpc/cca

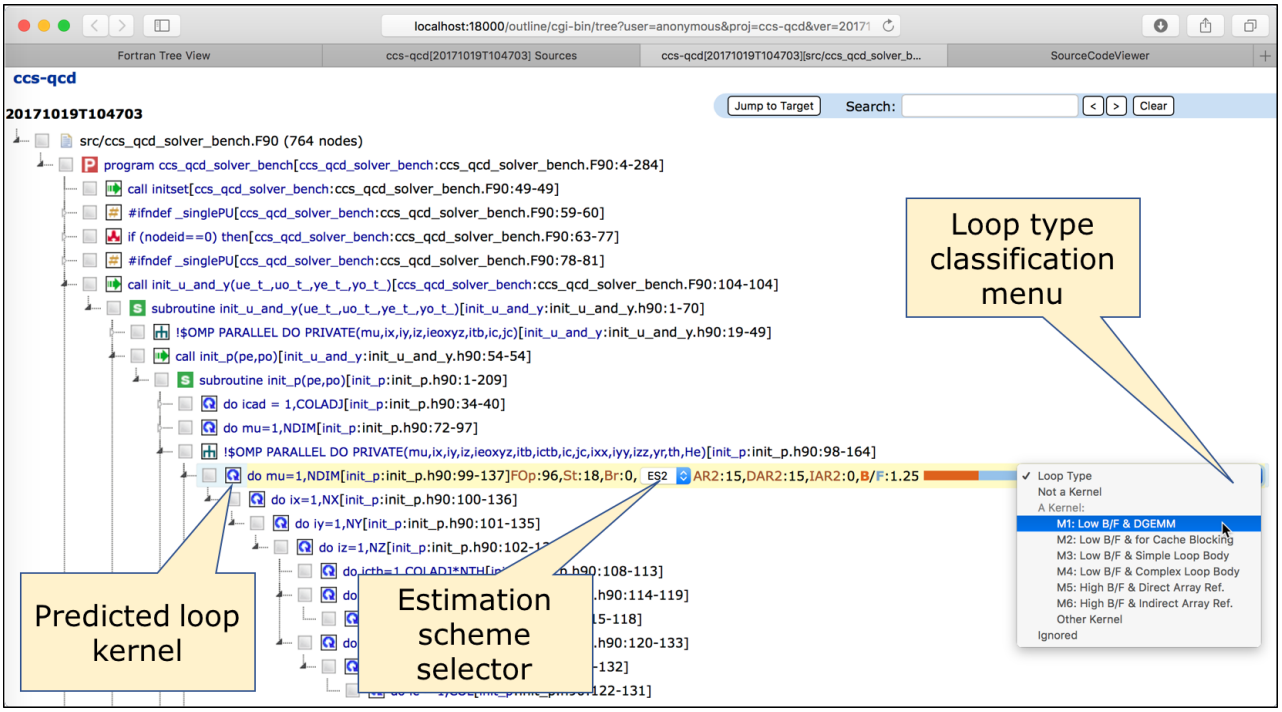

Figure 3: Tree view

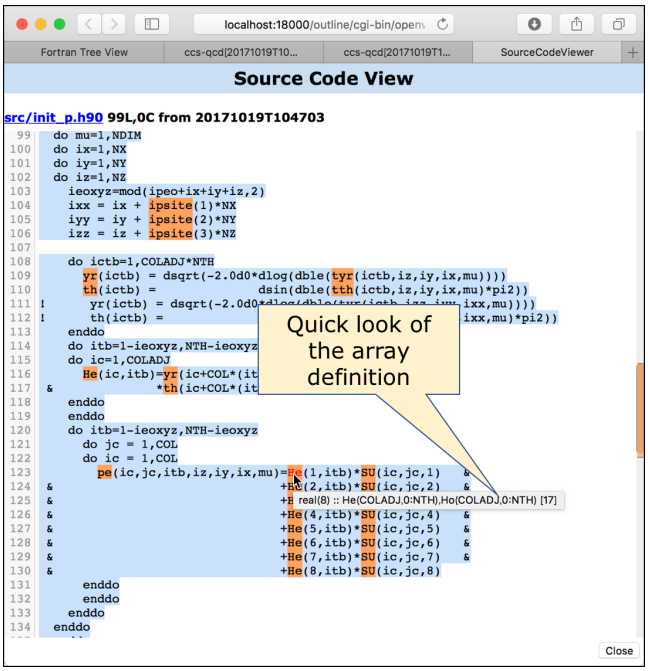

Figure 4: Source code view

ACKNOWLEDGMENTS

This work was supported in part by JSPS KAKENHI Grant Number JP26540031.

REFERENCES

[1] Protonu Basu, Mary Hall, Malik Khan, Suchit Maindola, Saurav Muralidharan, Shreyas Ramalingam, Axel Rivera, Manu Shantharam, and Anand Venkat. 2013. Towards Making Autotuning Mainstream. *International Journal of High Performance Computing Applications* 27, 4 (2013), 379–393.

[2] Forcheck b.v. [n. d.]. FORCHECK. ([n. d.]). http://www.forcheck.nl/.

[3] Chih-Chung Chang and Chih-Jen Lin. 2011. LIBSVM: A Library for Support Vector Machines. *ACM Trans. Intell. Syst. Technol.* 2, 3 (2011), 27:1–27:27.

[4] Scott Deerwester, Susan T. Dumais, George W. Furnas, Thomas K. Landauer, and Richard Harshman. 1990. Indexing by latent semantic analysis. *Journal of the American Society for Information Science* 41, 6 (1990), 391–407.

[5] Masatomo Hashimoto, Masaaki Terai, Toshiyuki Maeda, and Kazuo Minami. 2015. Extracting Facts from Performance Tuning History of Scientific Applications for Predicting Effective Optimization Patterns. In *Proceedings of the 12th IEEE/ACM Working Conference on Mining Software Repositories (MSR)*. 13–23.

[6] Masatomo Hashimoto, Masaaki Terai, Toshiyuki Maeda, and Kazuo Minami. 2017. An Empirical Study of Computation-Intensive Loops for Identifying and Classifying Loop Kernels. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering (ICPE)*. 361–372.

[7] Ralph Johnson, Jeff Overbey, and Greg Watson. [n. d.]. Photran -An Integrated Development Environment and Refactoring Tool for Fortran. ([n. d.]). http://www.eclipse.org/photran/.

[8] Dominic Orchard and Andrew Rice. 2013. Upgrading Fortran Source Code Using Automatic Refactoring. In *Proceedings of the 2013 ACM Workshop on Refactoring Tools (WRT)*. 29–32.

[9] Ananta Tiwari, Jeffrey K Hollingsworth, Chun Chen, Mary Hall, Chunhua Liao, Daniel J Quinlan, and Jacqueline Chame. 2011. Auto-tuning Full Applications: A Case Study. *Int. J. High Perform. Comput. Appl.* 25, 3 (2011), 286–294.

[10] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Commun. ACM* 52, 4 (2009), 65–76.