Extracting Facts from Performance Tuning History of Scientific Applications for Predicting Effective Optimization Patterns

Masatomo Hashimoto, Masaaki Terai, Toshiyuki Maeda and Kazuo Minami
RIKEN Advanced Institute for Computational Science
Kobe, Hyogo 650-0047, Japan
Emails: {m.hashimoto,teraim,tosh,minami_kaz}@riken.jp

***Abstract*—**

To improve performance of large-scale scientific applications, scientists or tuning experts make various empirical attempts to change compiler options, program parameters or even the syntactic structure of programs. Those attempts followed by performance evaluation are repeated until satisfactory results are obtained. The task of performance tuning requires a great deal of time and effort. On account of combinatorial explosion of possible attempts, scientists or tuning experts have a tendency to make decisions on what to be explored just based on their intuition or good sense of tuning. We advocate evidence-based performance tuning (EBT) that facilitates the use of database of facts extracted from tuning histories of applications to navigate the search space. However, in general, performance tuning is conducted as transient tasks without version control systems. Tuning histories may lack explicit facts about what kind of program transformation contributed to the better performance or even about the chronological order of the source code snapshots. For reconstructing the missing information, we employ a state-of-the-art fine-grained change pattern identification tool for inferring applied transformation patterns only from an unordered set of source code snapshots. The extracted facts are intended to be stored and queried for further data mining. This paper reports on experiments of tuning pattern identification followed by predictive model construction conducted for a few scientific applications tuned for the K supercomputer.

I. Introduction

For computational scientists that conduct large-scale scientific computations on supercomputers, application performance tuning is essential to make full use of the available computing resources and hence to maximize their scientific results [1]. To improve the performance (or execution time, code size, and power consumption) of application programs, scientists themselves or performance tuning experts make various empirical attempts to change compiler options, to change program parameters, or even to transform the programs without changing their semantics. Those attempts followed by performance evaluation are repeated until satisfactory results are obtained.

The process of performance tuning still remains more or less manual and requires a great deal of time and effort, although a number of studies on auto-tuning systems are conducted [2]. Auto-tuning systems rely on empirical techniques, which evaluate possible implementations of a computation to spot the best one in an automated manner. However, it suffers from combinatorial explosion of the search space. For example, approximately a hundred of flags for performance tuning are available in the GNU compiler collection (GCC) [3], which forces us to explore an extremely large search space of possible candidates.

Accordingly, scientists and/or tuning experts have a tendency to make decisions on what to be explored just based on their intuition or good sense of tuning. It is almost impossible to teach others about the intuition or the sense unlike knowledge of numerical algorithms, compiler optimizations, or computer architectures.

We advocate evidence-based performance tuning (EBT) after evidence-based medicine (EBM) [4]. While the original EBM emphasizes teaching the practice of medicine and improving decisions by individual physicians, EBT stresses teaching the practice of performance tuning and improving decisions by individual tuners. Just as EBM, EBT deprecates intuition, unsystematic experience, and computer scientific rationale as sufficient grounds for decision making in performance tuning. It facilitates the use of database of facts extracted from tuning histories of applications to navigate the search space.

In this study, we extract, from the given set of tuning histories, facts about what kind of program transformation (typically loop transformation) contributed to the better performance. To extract such facts, we expect at least the following data in a history:

1) a set of pairs each of which consists of a snapshot of source code, or a *source code variant*, and another variant derived directly from it,
2) program transformations applied between each pair of source code variants, and
3) performance measurement values for each code variant.

In general, however, performance tuning is conducted as dedicated transient tasks without version control systems. One or more of those data may be missing since scientists/tuners will not bother to record individual tuning details, which indicates that even the chronological order of the source code variants may be lost.

While the performance of code variants can be measured with profiling tools as long as the variants are compiled and executed, applied program transformations cannot be identified

978-0-7695-5594-2/15 © 2015 IEEE MSR2015, Florence, Italy

in a straightforward way. We employ a state-of-the-art fine-grained change pattern identification tool [5] for the purpose. It is capable of identifying pre-defined program transformation patterns applied to a code variant by comparing it with its derivative. Even when it is not known that which variant is derived from which, we can guess it by way of phylogenetic trees generated by applying phylogenetic algorithms to source code variants [6] instead of biological species or genes. Once sufficient facts are extracted, they are stored into a database called *factbase* and can be queried for further data mining to facilitate performance tuning, for instance, by predicting effective program transformation patterns.

To demonstrate that above mentioned techniques actually work for non-trivial scientific applications, we conducted experiments on a few scientific applications actually tuned for the world's fourth-fastest supercomputer (as of writing), the K computer [7], [8]. The experiments are summarized as follows:

- recovering phylogenetic relationships between code variants,
- identifying applied code transformation patterns, and
- constructing classifiers for predicting effective program tuning patterns for given source code.

Fortunately, we could access complete tuning histories of the applications by courtesy of the authors/tuners of them and hence we can evaluate our results by comparing them with the actuals.

The rest of the paper is organized as follows. Section II gives an overview of the fact extraction for EBT and its technical backgrounds. Then, factbase construction and factbase query are explained in Section III. A brief overview of the tuning pattern prediction is given in Section IV and Section V details experiments conducted for early-stage practice of EBT. After related work and several limitations are reviewed in Sections VI and VII, Section VIII concludes the work.

II. Fact Extraction

In this study, we focus on single processor performance tuning of scientific applications. In general, a task of performance tuning of application programs consists of the following steps:

1) measuring runtime performance,
2) identifying performance bottlenecks,
3) extracting *computational kernels*, and
4) performing the following for each kernel:
    a) measuring fine-grained runtime performance,
    b) diagnosing performance problems, and
    c) transforming source code of the kernel.

As is usual with performance tuning, compilable pieces of code that are subject to tuning, called *computational kernels*, or *kernels* for short, are extracted from the whole application source code. To diagnose performance problems in a kernel, more detailed and accurate performance measurements such as cache miss rate and floating-point operations per second for tuner-specified code regions (typically loops) are used. Based on the diagnosis, a tuner may perform some source code transformations (typically loop optimizations) on the kernel code in a manner that preserves the original semantics. Consequently, another *variant* of the kernel code is derived. Step (4) above is repeated until the desired performance is achieved, which results in a set of *variant derivation graphs* that are similar to revision graphs seen in some version control systems. A variant graph is a directed acyclic graph that consists of a set of edges $v_1 \rightarrow v_2$, where a variant v_2 is derived from another v_1.

In order to facilitate EBT, we should be able to elicit cause-and-effect relationships between applied source code transformations and the consequent performance gain (or loss) from factbases. Thus a factbase should contain at least the following information:

- kernel source code,
- performance data measured for the kernel, and
- a set of pairs each of which consists of a kernel and its variant with applied source code transformation.

However, we have learned through a preparatory study that it is not reasonable to expect tuning experts to have been using version control systems for performance tuning or recording detailed history of their manual tuning effort, and therefore that one or more items above might not be available. We employ a method of identifying transformation patterns applied to source code [5] assuming that at least a set of source code variants derived from an original kernel is available. For each kernel, the identification process is divided into the following steps:

1) Parsing source code variants to obtain their abstract syntax trees (ASTs).
2) Comparing the ASTs to obtain pairwise editorial distances.
3) Inferring a *phylogenetic tree* of code variants based on the distances.
4) Comparing (again) the ASTs based on the topology of the phylogenetic tree to infer fine-grained structural changes made to each code variant.
5) Storing the ASTs with the inferred fine-grained changes into a factbase.
6) Identifying transformation patterns by querying the factbase.

Note that steps (2) and (3) are not necessary if the variant derivation graph is known in advance.

A. Parsing Fortran Code

For writing large-scale scientific applications, Fortran, a pioneer of the high-level programming languages, is still actively used. After a few months survey of available Fortran parsers, we made up our mind to develop our own Fortran parser from scratch since they lack one or more of the following:

- capability of parsing dialects and language extensions made locally by compiler vendors such as IBM, PGI, and Intel,
- capability of parsing directives such as C preprocessor, OpenMP[1], XLF (IBM), and OCL (Fujitsu),

[1] http://openmp.org/

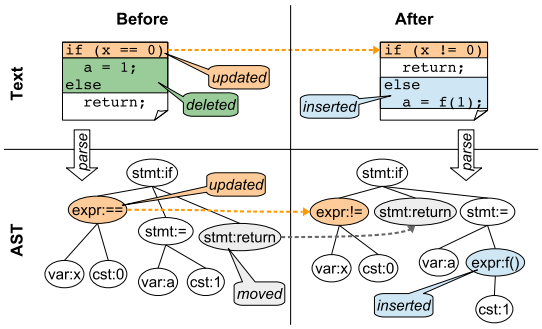

Fig. 1. Comparing Source Code

- tolerance for partial parsing failure, and
- capability of parsing incomplete program fragments.

Our parser is based on the several standards: FORTRAN 77, Fortran 90, Fortran 95 and in part Fortran 2003 and Fortran 2008. It is capable of parsing above mentioned language extensions and compiler directives. It can run in keep-on-parsing mode since we made use of Menhir[2], a LR(1) parser generator, with error recovery function enabled to build the core of the parser. It is also capable of parsing program fragments such as sequences of statements.

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 can be left ambiguous. For example, since both an array access and a function reference are written in a form like a(x), the type of a is necessary to disambiguate the entity a(x).

The parser is intensively tested for numerous applications including Gaussian[3] and EigenExa[4]. The number of tested source files amount to more than 20,000, the source lines of code (blank lines and comment lines excluded) more than 6,600,000, and the number of AST nodes more than 62,000,000. Although about 6% of the AST nodes are left ambiguous by our parser, we can disambiguate them later by resolving dangling references in a factbase.

B. Comparing ASTs

We prefer node-by-node comparison rather than line-by-line comparison because the former can give us more detailed information about the changes as seen in Figure 1. In the figure, a code snippet at the top left is modified to yield another at the top right. You will immediately see that line-by-line text comparison tells us almost nothing about the structural changes other than a few editorial operations on some *lines*. On the other hand, a series of changes made to an AST is represented as a sequence of editorial operations with additional move operation on AST *nodes*. It tells us far more than in the node-by-node case. A couple of ASTs at the bottom are obtained by parsing a couple of the snippets at the top, respectively. Note that AST nodes are enriched with their syntactic categories such as "statement" and "expression" while parsing. Then comparing the ASTs results in the following information about the structural changes:

- inversion of the condition-part of the if-statement,
- interchange of the then-part and the else-part, and
- insertion of a function call at the right hand side of an assignment.

We use a fine-grained change analysis tool Diff/TS that is based on fine-grained tree differencing on ASTs [6] for identifying structural changes made to code variants. Diff/TS calculates for a given pair of ASTs a sequence of editorial operations, called an *edit sequence*, that transforms an AST into another. Diff/TS is capable of detecting moves of connected node groups in addition to the basic edit operations: relabeling, deletion, and insertion of nodes. While calculating edit sequences, Diff/TS also computes dissimilarity or edit distance between a pair of ASTs based on the number of editorial operations contained in the edit sequence. See our previous paper [6] for more detail.

C. Generating Phylogenetic Tree of Code Variants

Suppose that we have a tuning history that consists of only an unordered and unstructured set of code variants derived from an original code. We can naively perform pairwise comparisons among the variants in order to extract transformation patterns by storing the fine-grained changes into a factbase, and then querying the factbase for the patterns. However, this strategy is obviously inefficient in terms of time and disk space. To reduce the size of a factbase and the number of the factbase queries, we infer the chronological precedence for each pair of the code variants in order to filter out the chronologically impossible pairs. We construct a phylogenetic tree rooted at the original kernel code [6] to infer the chronological order of the variants based on the tree topology.

We rely on *distance-matrix methods* to construct phylogenetic trees. Distance-matrix methods are originally used in phylogeny to produce a phylogenetic tree based on a matrix of pairwise genetic distances between biological species or genes. Since they depend only on "distances", we can apply them to code variants by using distances between ASTs instead of genetic distances. In this study, we employ an established phylogenetic distance-matrix method called *weighted least squares method* by Fitch and Margoliash [9].

Once we have obtained a distance matrix by pairwise comparison of ASTs, we feed the matrix to a phylogenetics software package that supports Fitch-Margoliash such as PHYLIP[5]. It will produce a phylogenetic tree in a conventional

[2]http://cristal.inria.fr/~fpottier/menhir/
[3]http://www.gaussian.com/
[4]http://www.aics.riken.jp/labs/lpnctrt/EigenExa_e.html
[5]http://evolution.genetics.washington.edu/phylip.html

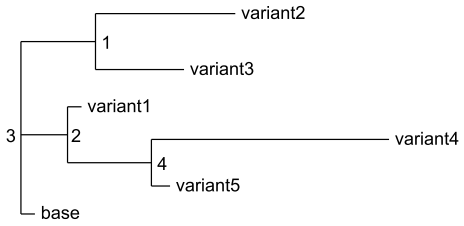

Fig. 2. A Phylogenetic Tree

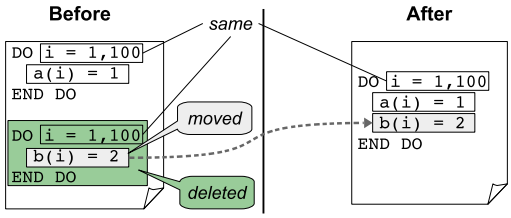

Fig. 3. A Code Transformation Pattern (Loop Fusion)

bifurcating form as seen in Figure 2, where the horizontal lines reflect the distances between tree nodes. The leaves correspond to code variants and the internal nodes (numbered 1 through 4) correspond to imaginary ancestor variants.

From the possible pairs of variants, we filter out pairs of variants (v_0, v_1) that satisfy the following condition:

$$\mathrm{distance}(mrca, v_0) \geq \mathrm{distance}(mrca, v_1),$$

where $mrca$ be the most recent common ancestor of v_0 and v_1, and $\mathrm{distance}(x, y)$ denotes the distance between x and y. This condition halves the variant pairs. For example, a variant pair $(\mathsf{variant4}, \mathsf{variant1})$ in the tree in Figure 2 is filtered out, because variant1 is closer to the most recent common ancestor "2". This means that variant1 likely precedes variant4.

D. Identifying Code Transformation Patterns

Since typical performance bottlenecks of scientific applications lie in loops, we mainly focus on identifying loop transformations. We give brief explanations for several basic loop transformations.

Loop Fission Loop fission divides a loop into multiple loops with the same iteration range. This transformation can reduce register spilling in a large loop body, while it can increase loop overheads. If the original loop has data dependencies between individual iterations, this can aid an optimizing compiler to perform software pipelining and to generate SIMD instructions by isolating the dependent part as another different loop.

Loop Fusion Loop fusion is the opposite of loop fission, which explicitly reduces loop overheads. Although this can improve temporal and spatial locality of data references, L1-data cache thrashing and register spilling arise according to the size of the combined loop body. The lack of available registers may also prevent an optimizing compiler from performing optimizations such as software pipelining.

Loop Unrolling Loop unrolling reduces the number of loop iterations by replicating the body of loops. Thus, it is effective for a loop whose body is too small to ignore the loop overheads. In addition, loop unrolling is also beneficial in the sense that it makes the body of loops larger, hence increases the chance of (automatic) vectorization and/or SIMD parallelization, with the trade-off between the increase and the risks of instruction cache overflow and register spills.
One important factor when applying loop unrolling is how many times should we replicate a loop body because it greatly affects the performance of the unrolled loops. It should be noted that, in this work, we do not try to find the optimal number of replication, but rather find loops that should be unrolled. Finding the optimal number can be solved by, for example, ordinary automatic performance tuning approaches, and it is beyond the scope of this work, although it seems possible to integrate this work with the approaches.

Array Merging Array merging merges two or more arrays of the same size into one single array. Array merging can reduce data cache misses if the arrays to be merged are accessed in a similar pattern in a loop, because it can greatly improve data access locality. In addition, array merging is also effective when considering (automatic) vectorization and/or SIMD parallelization.

Array Shape Change Array shape change permutes the dimensions of a multidimensional array in order to reduce data cache misses. Thus, array shape change may be effective when a multidimensional array is accessed with a stride that is larger than the size of data cache (and/or line) in a loop. It is also beneficial when considering vectorization and/or SIMD parallelization, in the same way as array merging.

We identify source code transformation patterns only from a pair of source code variants. Suppose that we have a base code variant and its derivative. In order to identify transformation patterns applied to the base code, we make use of the ASTs and an edit sequence obtained by comparing them. Figure 3 illustrates how "loop fusion" can be described in terms of AST and edit sequence. From the ASTs, we can conclude that there are two original loops in the left code snippet and there is a target loop in the right code snippet, where the three loop controls are the same. From the edit sequence, we can also conclude that there is an unchanged statement in one of the original loops, that one of the original loops is deleted, and that a statement in the deleted loop is moved to the target loop.

III. FACTBASE: DATABASE OF FACTS

We intend to make detailed information extracted from performance tuning efforts available in a way that is independent of specific tools, platforms, and programming languages to facilitate EBT. To achieve this, we make use of Semantic Web[6] technologies, which aim at describing, publishing, and obtaining relationships between things on the Web. We explain basic ideas of factbase construction and analysis by way of factbase queries in the following.

[6]http://www.w3.org/standards/semanticweb/

A. Facts and Ontologies

A fact about things in performance tuning is described as a triple of *subject*, *predicate* (also called *property*), and *object* following the Resource Description Framework[7] (RDF). Both subjects and objects may be entities in performance tuning such as performance data, applied transformations, and source code entities (e.g. files, functions/methods, and statements). A predicate/property denotes a binary relation between a couple of entities or between an entity and its attribute. In the latter case, objects may be literals. For example,

$$(e, \mathsf{name}, \texttt{"foo"})$$

represents a fact that an entity e has a name `foo`. Kinds of entities and predicates such as "variable" and "name" above are specified by what is called vocabularies or *ontologies*. Ontologies define concepts and relationships used for describing facts about performance tuning. We have designed the following ontologies using the OWL ontology language[8]:

PA An ontology for performance analysis, where subclasses of pa:PaData will be defined for independent performance analysis tools.

TUNE An ontology for performance tuning patterns that consists of loop transformations and others.

VER An ontology for versions. A code variant is designated by an instance of ver:Variant.

CHG An ontology for source code changes in terms of AST obtained by parsing source code.

SRC A core ontology for source code entities independent of specific programming languages.

FORTRAN An ontology for Fortran language that defines a subclass of src:TextEntity. The classes of Fortran entities are defined based on Fortran language specifications such as FORTRAN 77, Fortran 90, and other dialects.

In Figure 4, the hierarchy of conceptual classes of the ontologies is shown. In OWL, a class is defined as a subclass of owl:Thing. We disambiguate names of conceptual classes by prefixing namespaces to names as owl:Thing or src:Entity. We use namespaces listed in Table I throughout the rest of the paper.

Prefix	Meaning
rdf:	Resource Description Framework
rdfs:	RDF Schema
owl:	OWL Web Ontology Language
src:	Core source code entity
f:	Fortran source code entity
chg:	Things related to source code changes
ver:	Things related to versioning
pa:	Things related to performance analysis
t:	Things related to performance tuning

TABLE I
NAMESPACE PREFIXES

In addition to the classes, we also have defined predicates for each ontology. There exist two types of predicates in OWL:

[7]http://www.w3.org/RDF/

[8]http://www.w3.org/standards/techs/owl#w3c_all

Predicate	Domain	Range
f:inProgramUnit	f:Entity	f:ProgramUnit
⊢ f:inMainProgram	f:Entity	f:MainProgram
f:name	f:Entity	rdfs:Literal

TABLE II
PREDICATES FOR FORTRAN ONTOLOGY (EXCERPTS)

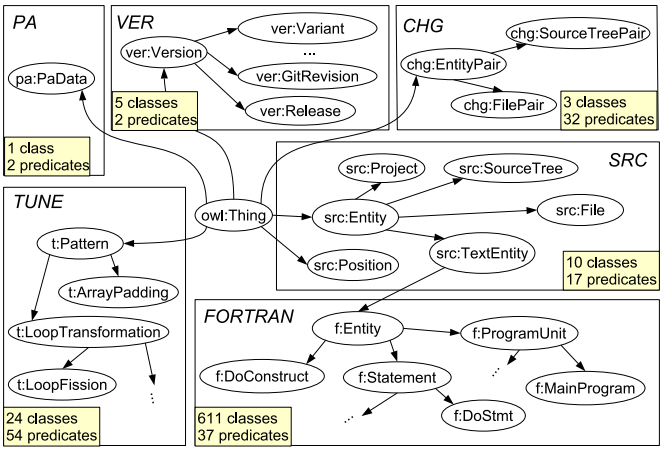

Fig. 4. Classes for Ontologies (Excerpts)

object properties, which are the relations between instances of conceptual classes, and *datatype properties*, which are relations between instances and RDF literals or possibly values of XML schema datatypes[9]. A predicate is defined in OWL as a subproperty of owl:ObjectProperty or owl:DatatypeProperty.

Table II shows excerpts from predicates defined in *FORTRAN*. The first two predicates are object properties and the rest is a datatype property. A predicate f:inMainProgram is a subproperty of f:inProgramUnit. The domain and the range of a predicate are also specified in OWL.

According to the RDF data model, a set of facts form a directed graph, where each triple is represented by a graph fragment $s \xrightarrow{p} o$. Figure 5 depicts an example of a fact graph, where p, c, and d denote source code entities that designate a *main-program*, a *do-construct* (or a loop), and a *do-stmt*, respectively. As seen in the graph, a predicate rdf:type is used to specify conceptual classes of entities. Note that a predicate src:parent from *SRC* defines parent-children relationships in terms of ASTs.

CHG provides predicates partially listed in Table III. Figure 6 illustrates facts of AST changes that constitute "loop

[9]http://www.w3.org/XML/Schema/

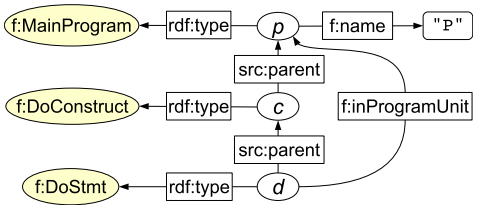

Fig. 5. A Fact Graph

Predicate	Domain	Range
chg:mappedTo	src:Entity	src:Entity
⊢ chg:mappedEqTo	src:Entity	src:Entity
⊢ chg:mappedNeqTo	src:Entity	src:Entity
chg:deletedOrPruned	src:Entity	src:Entity
⊢ chg:deletedFrom	src:Entity	src:Entity
⊢ chg:prunedFrom	src:Entity	src:Entity
chg:insertedOrGrafted	src:Entity	src:Entity
⊢ chg:insertedInto	src:Entity	src:Entity
⊢ chg:graftedOnto	src:Entity	src:Entity
chg:movedTo	src:Entity	src:Entity

TABLE III
PREDICATES FOR AST CHANGE ONTOLOGY (EXCERPTS)

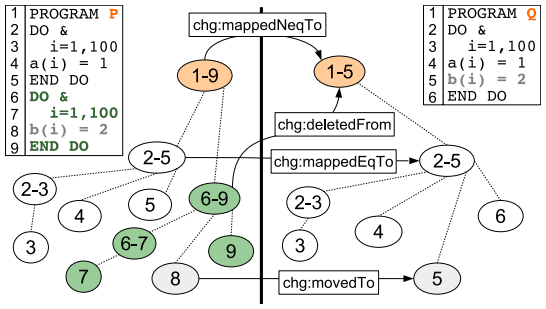

Fig. 6. Facts of AST Changes

fusion", where AST nodes are labeled with the corresponding line numbers or line number ranges and AST edges are indicated by dashed lines. Note that each of DO statements is continued by the "&" symbol to put its loop control on the next line. The meanings of the predicates shown in Table III are explained below in terms of AST changes, where examples are taken form Figure 6.

- A fact $(e, \text{chg:mappedTo}, e')$ means that a node e has a corresponding node e' in another AST. In a concrete term, e is either unchanged, relabeled or moved to be e'. When e is relabeled (e.g. `"P"`), it implies another fact $(e, \text{chg:mappedNeqTo}, e')$, otherwise it implies $(e, \text{chg:mappedEqTo}, e')$ (e.g. a subgraph that contains a pair of loop nodes).
- A fact $(e, \text{chg:deletedFrom}, e')$ means that a node e is deleted (e.g. the second loop in `P`) from its parent which corresponds to a node e' (e.g. 1-5) in another AST. The predicate chg:prunedFrom is used instead of chg:deletedFrom when a whole subtree rooted at e is deleted.
- A fact $(e, \text{chg:insertedInto}, e')$ means that a node e is inserted to be a child of a node which node e' in another AST corresponds to. The predicate chg:graftedOnto is used instead of chg:insertedInto when a subtree rooted at e is inserted.
- A fact $(e, \text{chg:movedTo}, e')$ means that a node e is moved to be e' (e.g. `b(i)=2`) in another AST.

B. Representing Source Code Entities

Once ontologies are defined, we create instances of the defined conceptual classes to describe facts about performance tuning. Since Semantic Web technologies use Internationalized Resource Identifiers (IRIs)[10] to identify things on the Web, we must associate an IRI with each entity. Among things related to performance tuning such as performance data, transformation patterns, and source code fragments, we here focus on source code entities.

As we aim to facilitate collaborative efforts in performance tuning, we are led to an idea of representing source code entities by textual regions of source files [10]. Since any tools and users can point code regions of their interests, it serves as universal and independent way of sharing and exchanging information about source code. To be concrete, we can associate an IRI with a source code entity by first concatenating and then encoding the followings:

- an ID of the source file in which the entity resides,
- the start position in the file, and
- the end position in the file,

for which we can employ a hash value to identify a file, and a triple of a line number, a column number and an offset to specify a position in the file. For example, an IRI

`http://example.com/fact/entity/FDLCO-MD5_ed9d31a4556c7ca7dbfcf99b31fb9ec5-5_0_34_5_5_39`

represents an entity (*end-do-stmt*) located between column 0 to 5 at line 5, and also between offset 34 to 39 in a source file of the left code in Figure 6 which has an MD5 hash value as encoded in the IRI.

C. Factbase Query

Conceptually, a factbase is a database filled with a set of facts and ontologies. There are a number of commercial or open source software systems for managing factbases with ontologies. An RDF store is a database system specialized for storing and managing the RDF data. In the following, we explain some important issues arising in querying the factbase.

In order to search factbases for change patterns, we write queries for the patterns in SPARQL[11]. SPARQL is a standard query language for searching graph patterns in RDF stores. Roughly speaking, SPARQL is an extension of SQL with graph patterns described by a set of triples with variables. For example, consider the following query that will enumerate names of all main programs in the factbase.

```
SELECT DISTINCT ?name WHERE {
  ?prog a f:MainProgram ;
        f:name ?name .
}
```

This query instructs the RDF store to find fact graphs matching the pattern

$$\text{f:MainProgram} \xleftarrow{\text{rdf:type}} \text{?prog} \xrightarrow{\text{f:name}} \text{?name}$$

and report values for specified variables. The query contains a graph pattern in the WHERE clause, where identifiers prefixed by "?" denote query variables. A graph pattern is essentially a set of triples written in the format

$$\textit{subject predicate object} .$$

[10]http://www.ietf.org/rfc/rfc3987.txt
[11]http://www.w3.org/TR/sparql11-query/

that may contain abbreviation symbol "a" for rdf:type. Consecutive triples that share a subject can also be written as

$$\begin{array}{lll} subject & predicate & object\ ; \\ & \vdots & \vdots \\ & predicate & object\ . \end{array}$$

Note that predicates and ontology classes are prefixed. Mappings from prefixed names to IRIs which appear in the head of normal SPARQL queries are omitted for brevity.

IV. Predicting Tuning Patterns

This section explains how we predict effective tuning patterns by means of statistical classification techniques based on factbases. In this work, we rely on *supervised learning* to classify a code fragment according to its potential tuning pattern that may improve its performance. Supervised learning is a kind of machine learning tasks that uses known set of examples called *training set* to construct predictive models. For predicting effective performance tuning, we make use of an actual result of performance tuning as an example data, that is, a pair of an extracted feature of a code fragment and the actual tuning pattern adopted to improve the performance of the code.

We represent a source code feature as an n-dimensional vector of real numbers, called *feature vector*. This means that we characterize a code fragment by n attributes. As an feature vector of a code fragment, we can employ performance data measured for the code fragment such as cache miss rate and/or code metrics such as the maximum number of loops in the code fragment. A tuning pattern, which is a set of m predefined primitive code transformation patterns, is represented as an m-dimensional vector of 0 or 1, called *tuning pattern vector*. A training set is a set of examples $\{(\mathbf{x}, \mathbf{t}) | \mathbf{x} \in \mathbb{R}^n, \mathbf{t} \in \{0,1\}^m\}$, where $\mathbf{x}$ and $\mathbf{t}$ are feature vector and tuning pattern vector, respectively. By applying a classification algorithm we obtain a classification function $c \in \mathcal{C} : \mathbb{R}^n \to \{0,1\}^m$ that predicts $\mathbf{c}$ from $\mathbf{x}$.

V. Experiments

In this section, we report on our early-stage practice of EBT applying the methods presented in the previous sections. As sample application programs, we could use the following:

- a set of 65 very small programs for examining basic loop optimization patterns, called Tuning Catalog,
- a kernel program, called MG, that is a component of the NAS Parallel Benchmarks [11] for evaluating parallel supercomputers, and
- a simulation software suite for a global cloud resolving model, called NICAM [12], [13], [14].

All of these were written in Fortran and actually tuned for the K computer [7], [8] by the software development team of RIKEN AICS including the second and the last authors. Brief descriptions of the applications are in order.

Tuning Catalog As a training set of our predictive analysis, we created Tuning Catalog, which is a set of small programs that represent various optimization approaches that appear in typical compiler (optimization) textbooks, including multithreading with OpenMP. More specifically, it consists of 65 optimization cases, and each case consists of an original source code and its optimized version. In addition, Tuning Catalog also records the detailed profiling data for every optimization case by executing the programs (with and without optimization) on the K computer.

MG The Multi-Grid (MG) method is a benchmark program based on three dimensional discrete Poisson equation from NAS Parallel Benchmarks [11]. It applies several spatial grids in discretization of the linear equation, and interpolates values between the grids to accelerate the convergence of solutions.

NICAM NICAM is a real world application of global weather/climate simulations [12], [13], [14]. It uses a nonhydrostatic and fully compressive equation system that is discretized on an icosahedral grid arrangement. NICAM consists of two main components. One is the fluid dynamical solver which consists of stencil operators. The other is a set of physical components including cloud micro physics and atmospheric radiation. In this work, we extracted 8 kernel programs from the two main components based on profiling results, and attempted to analyze their optimization histories.

Name	Variants	SLOC (avg.)	
Tuning Catalog	2×65	3089	(47)
MG	4	6957	(1739)
NICAM (diffusion)	20	7329	(366)
NICAM (div2rev)	30	21540	(718)
NICAM (divdamp3D)	18	5965	(331)
NICAM (divergence)	13	1823	(140)
NICAM (gradient)	10	3313	(331)
NICAM (nsw6)	9×3	27316	(2635)
NICAM (radiation)	3	14526	(4842)

TABLE IV
Sample Scientific Applications

Table IV shows the number of code variants and source lines of code (SLOC) computed by SLOCCount[12] for each sample application. In the table, names of the kernels are shown in parentheses for NICAM. Note that the whole program of MG and the programs of Tuning Catalog themselves are handled as kernels. For nsw6, we prepared three different settings of datasets and hard-wired parameters.

We performed factbase creation, code transformation pattern identification, and predictive model construction for the applications. Phylogenetic tree construction was also performed for kernels that contains more than a few code variants. Since two of the authors are core members of the team that conducted the tuning, we have "correct answers" for evaluating the results of the experiments.

All experiments described in the rest of this section were performed on a workstation with 8-core Intel Xeon processor (3.0 GHz) with 64GB RAM.

[12]http://www.dwheeler.com/sloccount/

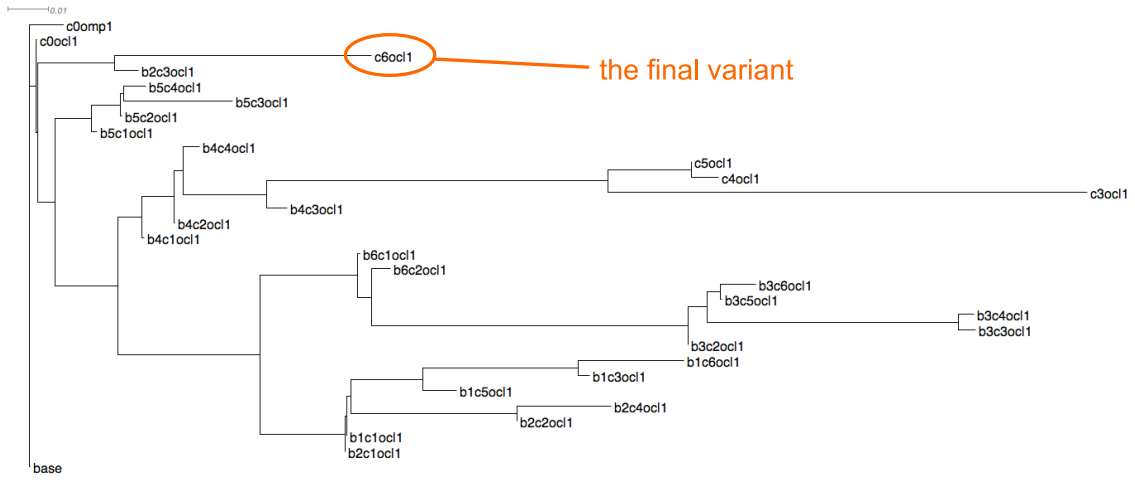

Fig. 7. A Phylogenetic Tree Generated for Variants of NICAM (div2rev)

Kernel	$\|R\|/\|P\|$	$\|R \cap T\|/\|T\|$
divergence	73/156 (0.47)	12/12 (1.00)
diffusion	187/380 (0.49)	21/21 (1.00)
gradient	41/90 (0.46)	9/9 (1.00)
nsw6	36/72 (0.50)	9/9 (1.00)
divdamp3D	153/306 (0.50)	21/21 (1.00)
div2rev	435/870 (0.50)	30/33 (0.91)

R: reduced pairs, P: all possible pairs, T: true pairs

TABLE V
REDUCED VARIANT PAIRS FOR NICAM KERNELS

A. Generating Phylogenetic Trees

We produced phylogenetic trees for NICAM kernels except radiation, which has only three variants, following the procedure mentioned in Section II-C. The required time varies from 1 minute (gradient) to 47 minutes (div2rev). A phylogenetic tree approximates a variant derivation graph. Figure 7 gives the largest tree among the NICAM kernels, where the final variant is circled. We can see, at a glance, that a great deal of efforts were made to tune div2rev, which turned out to be futile.

As shown in Section II-C, the generated phylogenetic trees could halve the possible parent-child variant pairs by giving chronological order. Table V summarizes the reduced set of pairs R, the set of all possible pairs P, and the set of true pairs T for each kernel. For the first three kernels, more than half of the possible pairs are filtered out, since there exist variant pairs each of which has components that are equally distant from the most recent common ancestor. The three false-negatives of div2rev were caused by a reversion in the variant derivation, which degrades the accuracy of the Fitch-Margoliash algorithm.

B. Creating Factbases

For all sample applications, the edit sequences obtained by comparing selected parent-child pairs were stored into an RDF store called Virtuoso[13], together with the following:

- the ontologies explained in Section III,
- the ASTs of the code variants,
- performance data of 55 measurement items measured for each measurement block with a dedicated profiling tool for the K computer, and
- 8 source code metrics for each measurement block,

where a measurement block is a code region delimited by a couple of special function calls. All the above were encoded into RDF triples. The resulting factbase contained 13,596,161 triples and occupied 460MB of disk space in total. It took about 15 minutes to store all the triples. We stored the edit sequences that come only from true variant derivation graphs in order to individually evaluate the rest of the experiments.

C. Identifying Code Transformation Patterns

We can identify code transformation patterns by querying the factbase. For example, we can write a query for loop fusion illustrated in Figure 3 (simplified for brevity):

```
SELECT DISTINCT ?loop0 ?loop1 ?loop_ WHERE {
    ?loop0 a f:DoConstruct ;
           f:loopControl ?lctl0  ;
           f:inProgramUnit ?pu ;
           chg:mappedTo ?loop_ .
    ?lctl0  src:treeDigest ?d0 .
    ?loop1 a f:DoConstruct;
           f:loopControl ?lctl1  ;
           f:inProgramUnit ?pu ;
           chg:prunedFrom ?e_ .
    ?lctl1  src:treeDigest ?d1 .
    FILTER (?loop1 != ?loop0 && ?d0 = ?d1)
},
```

where the equality of the loop controls is judged based on the subtree digest values. We wrote queries for 45 transformation patterns including loop fusion above and identified cumulative 891 effective optimization patterns for the sample applications. The 5 most frequent patterns were loop unrolling (8.53%), array dimension interchange (7.97%), loop interchange (7.07%), loop blocking (6.62%), and loop fusion (6.40%). The identified code transformation patterns were rendered as HTML files (Figure 8).

[13]http://virtuoso.openlinksw.com/dataspace/doc/dav/wiki/Main/

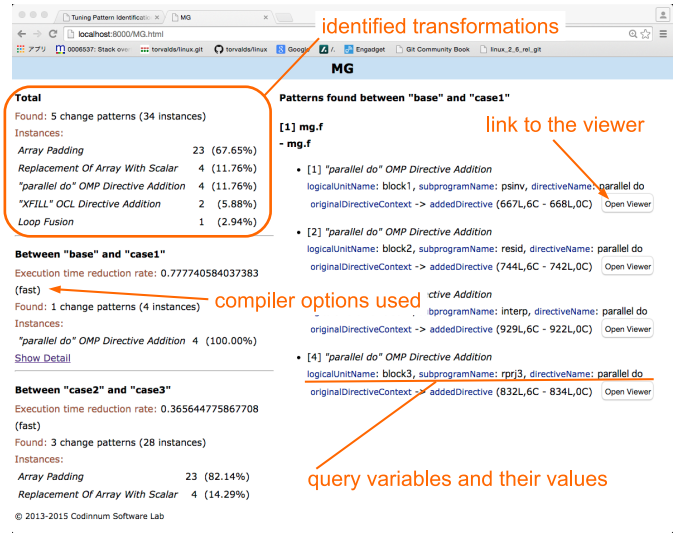

Fig. 8. Code Transformation Pattern Viewer

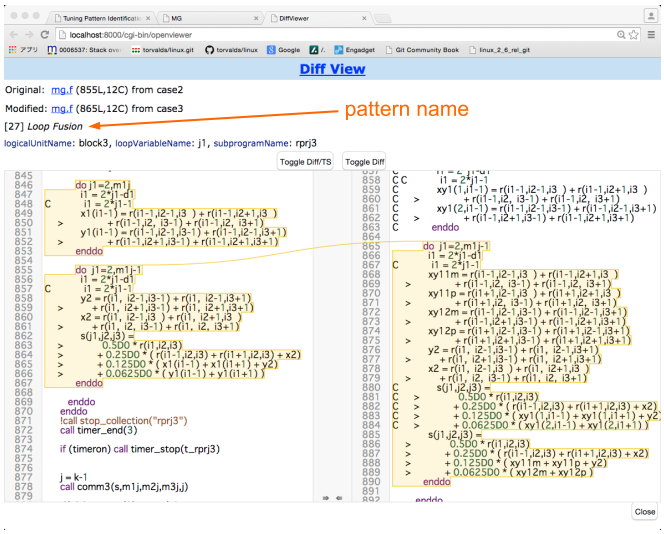

Fig. 9. Code Transformation Pattern Viewer (Simplified View)

By manually inspecting the identified patterns, we confirmed that we have identified all of the actual code transformations applied to the applications without any false positives. Inspecting a bunch of identified transformation patterns was a demanding task. To rapidly grasp the patterns, we have newly developed a simplified transformation pattern view (Figure 9) based on CodeMirror[14] in addition to the raw difference view (Figure 10).

D. Constructing Predictive Models

As explained in Section IV, we have to prepare a training set to construct a predictive model. We used the 55 measurement items and the 8 source code metrics mentioned in the previous section for feature vectors. A tuning pattern vector is a 45-dimensional vector that indicates whether each of the 45

[14]http://codemirror.net/

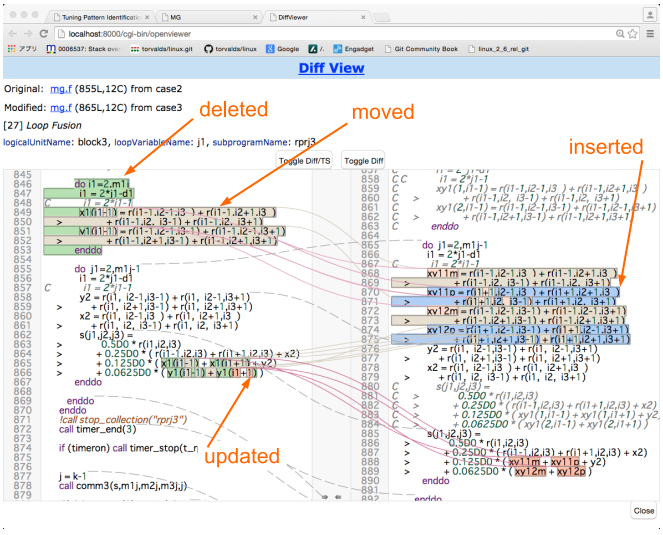

Fig. 10. Code Transformation Pattern Viewer (Raw Difference View)

transformation patterns is applied or not by 1 or 0. We extracted, by a simple query, a training set that consists of 469 instances each of which is a pair of the feature vector for a measurement block and a tuning pattern vector that improved the performance of the block.

We employed a data mining tool called Orange[15] to construct a predictive model from the training set by using BR method based on kNN. The *k nearest neighbors (kNN)* algorithm is one of the most basic machine learning algorithms for *single-label* classification. The learner just stores the training data, and the classifier makes predictions based on the k nearest neighbors of the data instance being classified. The *binary relevance (BR)* method is the most basic problem transformation method for *multi-label* classification [15]. Note that our tuning pattern vector has 45 different "labels" of transformation patterns. BR transforms multi-label classification problems into a set of single-label classification problems to perform multi-label classification based on single-label learning algorithms such as kNN.

The result of 5-fold cross validation of the model ($k = 10$) is summarized as follows: *Brier score* is 0.047 (the lower the better), *global accuracy* is 0.510, and *mean accuracy* is surprising 0.970. Suppose that we have N instances of m-dimensional tuning pattern vectors. Let $\mathbf{p}_i$ and $\mathbf{a}_i$ be the i-th predicted and actual tuning pattern vectors, respectively. We write x_{ij} for the j-th component of the i-th vector $\mathbf{x}_i$. Brier score, or *mean squared error*, is defined as the average of the sum of errors over the instances: $\frac{1}{N}\sum_{i=1}^{N}(\mathbf{p}_i - \mathbf{a}_i)^2$. Global accuracy [16] is defined as per-instance accuracy: $\frac{1}{N}\sum_{i=1}^{N}\delta(\mathbf{p}_i, \mathbf{a}_i)$, where $\delta(\mathbf{x}, \mathbf{y})$ equals to 1 if $\mathbf{x} = \mathbf{y}$ or 0 otherwise. Mean accuracy [16] is defined as per-label accuracy: $\frac{1}{m}\sum_{j=1}^{m}\frac{1}{N}\sum_{i=1}^{N}\delta(p_{ij}, a_{ij})$, where $\delta(x, y)$ equals to 1 if $x = y$ or 0 otherwise.

[15]http://orange.biolab.si/

The result implies that the model is promising for suggesting effective source code transformation patterns, although it is not practical for predicting *exact* sets of effective patterns.

VI. Related Work

As briefly explained in Section I, it is almost impossible in general to completely automate performance tuning processes. Nevertheless several studies have been made on supporting performance tuning processes. Milepost GCC [3] is the first open-source machine learning based compiler. Based on the observation that similar programs may require similar optimizations, Milepost GCC correlate static program features and compiler optimizations to predict good compiler optimization flags for unseen programs. It employs a public repository to record compilation and execution statistics, which are later used as training data for the machine learning models.

Chaimov and others [17] automated collection of performance data annotated with metadata identifying properties of the execution environment and the input data by integrating TAU Performance System [18] and auto-tuning tools. They reported how annotated performance data can be used to learn classifiers which can be used for runtime selection of specialized function variants and for reducing the number of evaluations necessary for auto-tuning.

PerfExpert [19] is a tool for detecting and diagnosing performance bottlenecks. Recently, Fialho and others [20] enhanced it to provide suggestions for bottleneck remediation for given applications. The rules to select and rank recommendations are implemented as SQL queries which make use of metrics and features of the source code of the applications. These recommendations can be a general modification recipe for source code or suggestions to add compiler flags.

Grebhahn and others [21] applied an approach of optimizing software product lines [22] to optimize stencil computation which is extensively used in scientific computations. Because stencil computation has a lot of configuration options and tuning parameters, it is extremely hard to automatically calculate the optimal choice owing to combinatorial explosion. They utilized SPL Conqueror [22] to predict the (nearly) optimal choice by measuring performance of a small number of combination of configuration options and tuning parameters. One major difference from our work is that their target is limited to stencil computation, while our work does not assume a specific computing method.

There is also a number of studies concerning change pattern identification in the context of code refactoring [23]. Demeyer and others proposed a heuristic method based on metrics about entities such as method/class sizes and numbers of inherited/overwritten methods [24] to mine refactoring patterns by comparing two versions of the program. Although it is one of the earliest attempt at automated change pattern detection, the method is vulnerable to renaming, unable to discriminate multiple patterns in the same piece of code, and difficult to fine-tune.

Prete and others proposed a refactoring reconstruction tool called Ref-Finder [25]. Ref-Finder identifies refactoring patterns between a pair of programs written in Java. Ref-Finder expresses each refactoring pattern in terms of template logic rules based on the facts extracted from the programs and uses a logic programming engine to infer concrete refactoring instances. However, information at the granularity of expressions are lost during the process of fact extraction and hence fine-grained changes, concerning local variables for instance, can not be easily detected.

VII. Limitations

This section discusses several limitations of the methods presented in the paper.

Since we currently employ a phylogenetic tree construction algorithm as it is, the result for a set of code variants is inherently limited to a "tree", although actual code variants can be derived more than one parent forming a "graph". In fact, some of the true variant derivation graphs of the NICAM kernels (diffusion, divdamp3D, div2rev, and nsw6) contain edges caused by multiple inheritance. For such cases, phylogenetic network construction algorithms [26] might be applicable.

For identification of source code transformation, identifiable patterns are limited to known ones that can be described as factbase queries. Thus, it is possible to overlook unexpected variations. We might be able to obtain unknown transformation patterns by mining more source code repositories [27].

Another threat to the successful identification of code transformations is tangled change [28], which can also easily lower the recall rate of the identification. Although we kept conditions in the queries as weak as possible, untangling techniques [29] might handle the situation better.

While the constructed predictive model was unexpectedly accurate in terms of cross-validation results, it is not based on a satisfactory amount of examples. We are planning to extract more facts from public domain repositories of scientific applications even though they are not dedicated tuning histories.

VIII. Conclusions

We advocate in this study evidence-based performance tuning (EBT) of scientific applications to systematically support demanding tuning tasks. To embody the concept of EBT, we employed methods of analyzing source code repositories for inferring fine-grained source code changes, constructing phylogenetic trees for versions of source code, and identifying source code change patterns based on factbase queries.

To adapt the methods for performance tuning of scientific applications, we have developed a tolerant Fortran parser, ontologies for Fortran programs, queries for identifying code transformation patterns, and a neat viewer for the identified transformation patterns.

The results of experiments of constructing performance tuning factbases from minimum data and constructing prediction models for effective tuning patterns based on the factbases indicate that our approach is promising for practicing EBT.

Acknowledgment

This work was supported in part by JSPS KAKENHI Grant Number 26540031.

REFERENCES

[1] V. R. Basili, J. C. Carver, D. Cruzes, L. M. Hochstein, J. K. Hollingsworth, F. Shull, and M. V. Zelkowitz, "Understanding the high-performance-computing community: A software engineer's perspective," *IEEE Software*, vol. 25, no. 4, pp. 29–36, 2008.

[2] P. Basu, M. Hall, M. Khan, S. Maindola, S. Muralidharan, S. Ramalingam, A. Rivera, M. Shantharam, and A. Venkat, "Towards making autotuning mainstream," *International Journal of High Performance Computing Applications*, vol. 27, no. 4, pp. 379–393, 2013.

[3] G. Fursin, Y. Kashnikov, A. W. Memon, Z. Chamski, O. Temam, M. Namolaru, E. Yom-Tov, B. Mendelson, A. Zaks, E. Courtois, F. Bodin, P. Barnard, E. Ashton, E. V. Bonilla, J. Thomson, C. K. I. Williams, and M. F. P. O'Boyle, "Milepost GCC: machine learning enabled self-tuning compiler," *International Journal of Parallel Programming*, vol. 39, no. 3, pp. 296–327, 2011.

[4] G. Guyatt, J. Cairns, D. Churchill *et al.*, "Evidence-based medicine: A new approach to teaching the practice of medicine," *JAMA*, vol. 268, no. 17, pp. 2420–2425, 1992.

[5] M. Hashimoto, A. Mori, and T. Izumida, "A comprehensive and scalable method for analyzing fine-grained source code change patterns," in *Proceedings of the 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER'15)*, 2015, pp. 351–360.

[6] M. Hashimoto and A. Mori, "Diff/TS: A tool for fine-grained structural change analysis," in *Proceedings of the 15th Working Conference on Reverse Engineering (WCRE'08)*, 2008, pp. 279–288.

[7] M. Yokokawa, F. Shoji, A. Uno, M. Kurokawa, and T. Watanabe, "The k computer: Japanese next-generation supercomputer development project," in *Proceedings of the 2011 International Symposium on Low Power Electronics and Design (ISLPED)*, 2011, pp. 371–372.

[8] H. Miyazaki, Y. Kusano, H. Okano, T. Nakada, K. Seki, T. Shimizu, N. Shinjo, F. Shoji, A. Uno, and M. Kurokawa, "K computer: 8.162 petaflops massively parallel scalar supercomputer built with over 548k cores," in *2012 IEEE International Solid-State Circuits Conference (ISSCC) Digest of Technical Papers*, 2012, pp. 192–194.

[9] W. M. Fitch and E. Margoliash, "Construction of phylogenetic trees," *Science*, vol. 155, pp. 279–284, 1967.

[10] M. Hashimoto and A. Mori, "Enhancing history-based concern mining with fine-grained change analysis," in *Proceedings of the 16th European Conference on Software Maintenance and Reengineering (CSMR2012)*, 2012, pp. 75–84.

[11] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga, "The NAS parallel benchmarks—summary and preliminary results," in *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing (SC'91)*, 1991, pp. 158–165.

[12] H. Tomita and M. Satoh, "A new dynamical framework of nonhydrostatic global model using the icosahedral grid," *Fluid Dynamics Research*, vol. 34, no. 6, pp. 357–400, 2004.

[13] M. Satoh, T. Matsuno, H. Tomita, H. Miura, T. Nasuno, and S. Iga, "Nonhydrostatic icosahedral atmospheric model (NICAM) for global cloud resolving simulations," *Journal of Computational Physics*, vol. 227, no. 7, pp. 3486–3514, 2008.

[14] M. Satoh, H. Tomita, H. Yashiro, H. Miura, C. Kodama, T. Seiki, A. Noda, Y. Yamada, D. Goto, M. Sawada, T. Miyoshi, Y. Niwa, M. Hara, T. Ohno, S.-i. Iga, T. Arakawa, T. Inoue, and H. Kubokawa, "The non-hydrostatic icosahedral atmospheric model: Description and development," *Progress in Earth and Planetary Science*, vol. 1, no. 1, p. 18, 2014.

[15] G. Tsoumakas and I. Katakis, "Multi-label classification: An overview," *International Journal of Data Warehousing and Mining*, vol. 3, no. 3, pp. 1–13, 2007.

[16] J. H. Zaragoza, L. E. Sucar, E. F. Morales, C. Bielza, and P. Larrañaga, "Bayesian chain classifiers for multidimensional classification," in *Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI'11) - Volume Three*, 2011, pp. 2192–2197.

[17] N. Chaimov, S. Biersdorff, and A. D. Malony, "Tools for machine-learning-based empirical autotuning and specialization," *International Journal of High Performance Computing Applications*, vol. 27, pp. 403–411, 2013.

[18] S. S. Shende and A. D. Malony, "The tau parallel performance system," *International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 287–311, 2006.

[19] M. Burtscher, B.-D. Kim, J. Diamond, J. McCalpin, L. Koesterke, and J. Browne, "PerfExpert: An easy-to-use performance diagnosis tool for hpc applications," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC'10)*, 2010, pp. 1–11.

[20] L. Fialho and J. Browne, "Framework and modular infrastructure for automation of architectural adaptation and performance optimization for HPC systems," in *Proceedings of the 29th International Supercomputing Conference (ISC'14)*, 2014, pp. 261–277.

[21] A. Grebhahn, S. Kuckuk, C. Schmitt, H. Köstler, N. Siegmund, S. Apel, F. Hannig, and J. Teich, "Experiments on optimizing the performance of stencil codes with spl conqueror," *Parallel Processing Letters*, vol. 24, no. 03, 2014.

[22] N. Siegmund, S. S. Kolesnikov, C. Kästner, S. Apel, D. Batory, M. Rosenmüller, and G. Saake, "Predicting performance via automated feature-interaction detection," in *Proceedings of the 34th International Conference on Software Engineering (ICSE2012)*, 2012, pp. 167–177.

[23] M. Fowler, *Refactoring: Improving the Design of Existing Code (Addison-Wesley Object Technology Series)*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[24] S. Demeyer, S. Ducasse, and O. Nierstrasz, "Finding refactorings via change metrics," in *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA'00)*, 2000, pp. 166–177.

[25] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim, "Template-based reconstruction of complex refactorings," in *Proceedings of the 26th IEEE International Conference on Software Maintenance (ICSM'10)*, 2010, pp. 1–10.

[26] D. H. Huson and C. Scornavacca, "A survey of combinatorial methods for phylogenetic networks," *Genome Biology and Evolution*, vol. 3, pp. 23–35, 2011.

[27] S. Negara, M. Codoban, D. Dig, and R. E. Johnson, "Mining fine-grained code changes to detect unknown change patterns," in *Proceedings of the 36th International Conference on Software Engineering (ICSE2014)*, 2014, pp. 803–813.

[28] K. Herzig and A. Zeller, "The impact of tangled code changes," in *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR'13)*, 2013, pp. 121–130.

[29] M. Dias, A. Bacchelli, G. Gousios, D. Cassou, and S. Ducasse, "Untangling fine-grained code changes," in *Proceedings of the 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER'15)*, 2015, pp. 341–350.