An Empirical Study of Computation-Intensive Loops for Identifying and Classifying Loop Kernels

[Full Research Paper]

Masatomo Hashimoto
RIKEN Advanced Institute for Computational Science
Kobe, Hyogo 650-0047, Japan
m.hashimoto@riken.jp

Masaaki Terai
RIKEN Advanced Institute for Computational Science
Kobe, Hyogo 650-0047, Japan
teraim@riken.jp

Toshiyuki Maeda
RIKEN Advanced Institute for Computational Science
Kobe, Hyogo 650-0047, Japan
tosh@riken.jp

STAIR Lab
Chiba Institute of Technology
Narashino, Chiba 275-0016, Japan
tosh@stair.center

Kazuo Minami
RIKEN Advanced Institute for Computational Science
Kobe, Hyogo 650-0047, Japan
minami_kaz@riken.jp

ABSTRACT

The process of performance tuning is time consuming and costly even if it is carried out automatically. It is crucial to learn from the experience of experts. Our long-term goal is to construct a database of facts extracted from specific performance tuning histories of computation-intensive applications such that we can search the database for promising optimization patterns that fit a given kernel.

In this study, as a significant step toward our goal, we explored a thousand computation-intensive applications in terms of the distribution of kernel classes, each of which is related to expected efficiency and specific tuning patterns. To statistically estimate the distribution of the kernel classes, 100 loops were randomly sampled and then manually classified by experienced performance engineers. The result indicates that 50–70% of the kernels are memory-bound and hence difficult to run efficiently on modern scalar processors. In addition, based on the classification results, we constructed experimental classifiers for identifying loop kernels and for predicting kernel classes, which achieved cross-validated classification accuracy of 81% and 65%, respectively.

Keywords

Software performance tuning; computation-intensive application; kernel classification; kernel prediction; Fortran parser

1. INTRODUCTION

For scientists that conduct large-scale computations on supercomputers, application performance tuning is essential to maximizing their scientific results. To improve the performance of an 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 the given computing system; attempts to change compiler options, to adjust program parameters, or even to transform the programs in a semantics-preserving manner are repeated until sufficient simulated time/resolution and sufficient accuracy of the physical models/algorithms are achieved [3].

While performance tuning is still a demanding manual task relying on experience and intuition in general, a number of studies on auto-tuning systems are conducted for some specific kernels such as stencil code, linear algebra solvers, and matrix multiplications [4], or even full applications [21]. However, the process of performance tuning is time consuming and costly even if it is carried out automatically. Since auto-tuning systems rely on empirical techniques that evaluate possible parameters and/or implementations of a computational kernel to spot the best one in an automated manner, combinatorial explosion of the search space is inevitable. For example, approximately a hundred of flags for performance tuning are available in the GNU compiler collection (GCC) [11], which forces us to explore an extremely large search space of possible candidates.

Thus, it is crucial to learn from the experience of performance tuning experts in a way that enables us to improve the performance of an application based on how they improved the performance of similar applications. In our previous work [14], we addressed the problem of extracting facts from performance tuning histories of computational kernels. As a proof-of-concept, we have created a database of facts, or *factbase*, extracted from performance tuning histories of several kernels of a few real world scientific applications. We extracted facts about loop transformations performed on an original kernel by comparing it with its variants made by experts, and stored the facts into a factbase together with a set of source code metrics and performance profiling data as the evidence.

Our long-term goal is to create a sufficient factbase that

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ICPE'17, April 22-26, 2017, L'Aquila, Italy

© 2017 ACM. ISBN 978-1-4503-4404-3/17/04…$15.00

DOI: http://dx.doi.org/10.1145/3030207.3030217

can be searched for promising optimization patterns, or even for patches, that match a given kernel. However, it is not an easy task to collect performance tuning examples suitable for factbase construction. In general, detailed histories of performance tuning of a large-scale scientific application tend to be lost, since the tuning process is temporary and independent of the development of the application. We should select tuning examples according to some criteria in order to efficiently expand a factbase.

Classifying kernels into several classes and examining the distribution of the classes over computation-intensive applications would guide us in seeking out performance tuning instances that should be supplied to the factbase preferentially. HPC Challenge Benchmark [8] characterizes computational kernels based on spatial and temporal locality in memory access streams and provides 7 representative benchmarks for high-performance computing (HPC) systems. Instead of traditional benchmarks, Asanovíc and others [1] introduced 13 *Dwarfs*, each of which is an algorithmic method that captures a pattern of computation and communication. However, to the best of the authors' knowledge, there is no large-scale survey in terms of the distribution of such classes over computation-intensive applications.

In this study, we introduce another set of kernel classes and explore a thousand computation-intensive applications in terms of the distribution of the kernel classes. The set of classes was proposed to estimate expected efficiency of kernels and has been used for drawing up plans of the performance tuning of scientific applications running on the K computer [23].

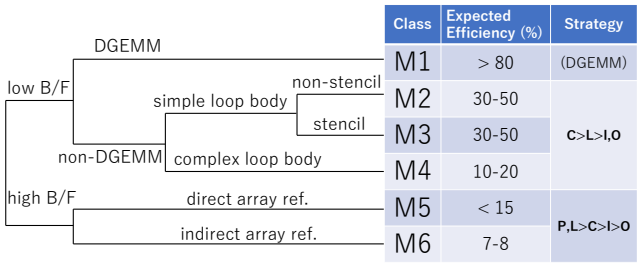
In order to statistically estimate the distribution of the kernel classes, 100 loops were randomly sampled from the 1000 applications and then manually classified by experienced performance engineers. The experts rely on static information extracted from the source code in addition to their experience and intuition. To support the classification task, we developed a web application that shows static features obtained by analyzing the source code.

In addition to the loop classification, we also discuss the correlation between static features of a kernel and its class by making use of the source code features and the manual classification results. We make an attempt to construct a binary classifier for predicting kernels and a multi-class classifier for predicting kernel classes by means of a supervised machine learning algorithm.

In summary, the aim of the study is two-fold:

- to reveal the distribution of the kernel classes over a thousand compute-intensive applications, and
- to examine the possibility of determining the class of a kernel only from static source code features of the kernel.

The remainder of the paper is organized as follows. Section 2 introduces the set of kernel classes. Then, static source code features that give supplementary information for manual kernel classification are presented in Section 3. After an overview of the analysis we carried out for extracting the source code features is given in Section 4, Section 5 details experiments conducted for a large-scale exploration of computation-intensive applications and for an examination of kernel and kernel class prediction. After related work and a few limitations are reviewed in Sections 6 and 7, Section 8 concludes the work.



Class	Expected Efficiency (%)	Strategy
M1	> 80	(DGEMM)
M2	30-50	C>L>I,O
M3	30-50	
M4	10-20	
M5	< 15	P,L>C>I>O
M6	7-8	

Figure 1: Kernel classes

2. CLASSIFYING LOOP KERNELS

To improve the performance of a scientific application, particular pieces of code, called *computational kernels*, or *kernels* for short, are extracted from it first. Kernels are identified by inspecting source code and/or by finding performance bottlenecks [20]. In this study, we focus on single-processor performance of a *loop kernel*, or a code region composed of one or several nested loops. We focus on the following in particular:

1. setting a performance target for each kernel, and
2. developing an optimization strategy for each kernel.

Once kernels are identified, a performance target is set and plans of optimization are drawn up for each kernel.

2.1 Classes of Computational Kernels

For the purpose of facilitating the steps above, we consider the *kernel classes* M1 through M6 shown in Figure 1. Each of the classes is related to expected efficiency, or a percentage of the theoretical peak performance (floating-point operations per second), and a typical tuning strategy. In the figure, the classes are arranged in descending order of expected efficiency, where rough estimates (%) for SPARC64VIIIfx, used in the K computer, are shown. Although efficiency percentages of M5/M6 kernels can be estimated based on the roofline model [22], typical values are shown in the figure. Note that the order of the classes should be similar for other modern scalar processors, although individual percentages may vary.

A tuning strategy is composed of the following basic steps, where examples of the relevant optimization techniques are shown in brackets for each step.

P Hiding memory access latency. [enabling prefetching]

L Exploiting data in cache lines. [array merging, data reordering]

C Improving temporal locality of memory accesses. [loop blocking]

I Improving instruction scheduling. [loop fission (to simplify loop bodies)]

O Utilizing fused multiply-add (FMA) and single instruction multiple data (SIMD) operations. [loop interchange (to place SIMD vectorizable loops innermost)]

Note that Figure 1 also shows the priority of these steps for each class except M1 in the figure. For example, "P,L>C" means that data prefetching and exploitation of cache lines

have priority over improving temporal locality. For further details on performing these steps, see the book [2] by Bailey and others for instance. The strategy for M1 is simply using DGEMM library as explained later in the next section.

2.2 Classification of Kernels

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. Note that we can also use multiplicative inverse of B/F, or *operational intensity*, used in the roofline model as long as it is used consistently.

In general, computations that require low B/F can be executed more efficiently than those that require high B/F, since the former can take advantage of cache memories. Thus, kernels are roughly divided into two classes: low-B/F kernels and high-B/F kernels. Then, each class is further divided into 4 (M1–4) or 2 (M5–6) classes based on the expected efficiency and the relevant optimization strategy.

We distinguish throughout the rest of the paper abstract B/F, denoted by $\mathcal{B}/\mathcal{F}$, essentially required by the computation that a kernel describes, from concrete B/F, denoted by $\mathsf{B/F}$, calculated by syntactically counting memory references and floating-point operations in kernel source code. As a rule of thumb, we expect less than 0.5 for low $\mathcal{B}/\mathcal{F}$ and more than 1.5 for high $\mathcal{B}/\mathcal{F}$.

2.2.1 Low $\mathcal{B}/\mathcal{F}$ Kernels

Some kernels that require low $\mathcal{B}/\mathcal{F}$ are essentially matrix-matrix multiplication and hence can be replaced with calls to an efficient matrix-matrix multiplication subroutine like DGEMM. They are classified as M1, where typical applications include first principles calculations based on density functional theory (DFT).

Otherwise, instead of DGEMM, cache blocking techniques may be effective in reducing the volume of memory traffic. If this is the case, the bodies of the loop kernels tend to be relatively simple in terms of size, the number of branching statements, and so on. They belong to M3 if they are stencil code and to M2 otherwise. M3 kernels have been rarely seen, while they can be executed relatively efficiently. Some particular high-order accuracy stencil computations are typical applications of M3 kernels. M2 kernels are used in, for example, classical molecular dynamics (MD) simulations of Coulomb interaction and n-body simulations of gravitational interaction.

If cache blocking is not effective, loop kernels tend to have relatively complex loop bodies and are considered to be in M4, where complex memory access patterns hinder efficient instruction scheduling and SIMD. Typical applications of M4 kernels include physical components of climate model simulations and plasma simulations based on particle-in-cell method.

2.2.2 High $\mathcal{B}/\mathcal{F}$ Kernels

Kernels that require high $\mathcal{B}/\mathcal{F}$ are simply divided into two classes depending on whether they have indirect array references or not. If a kernel does not contain indirect array references, it is in M5. M5 kernels are often seen in standard stencil computations. Typical applications include dynamical components of climate model simulations, fluid dynamics simulations, and earthquake simulations.

	Feature	Judgment
1	AR (Array References)	M1–4 | M5–6
2	FOp (Floating-point Ops)	(by $\mathsf{B/F}$)
3	St (Statements)	
4	Br (Branches)	
5	Ca (Procedure Calls)	M2–3 | M4
6	MAR (Max. Array Rank)	(by complexity)
7	MLD (Max. Loop Depth)	
8	IAR (Indirect Array References)	M5 | M6
9	MLL (Max. Loop Nest Level)	
10	MFL (Max. Fusible Loops)	kernel | non-kernel
11	MMA (Max. Mergeable Arrays)	

Table 1: Syntactic features of a loop

Kernels in the last class, M6, use indirect array references. Typical applications include structural simulations and fluid dynamics simulations based on finite element method (FEM).

3. STATIC FEATURES OF LOOPS

3.1 Loop Features

As explained in Section 2, we set a performance target for each kernel at an early stage of performance tuning. This is done by inferring the kernel class; we rely in part on syntactic features extracted from the source code for the inference. Table 1 presents our proposal for a set of source code features we consult to narrow down the candidate classes.

The first two features are for estimating $\mathsf{B/F}$ of a kernel and hence used for distinguishing M1–4 from M5–6. Note that $\mathsf{B/F}$ of a loop may differ from its $\mathcal{B}/\mathcal{F}$, since multiple implementations are possible for a computation. A kernel classification task requires comprehensive interpretation of data including $\mathsf{B/F}$ values, comment lines, and supplemental documents. The third through seventh are for measuring the complexity of a loop body. They are used to distinguish M2–3 and M4. The eighth feature distinguishes M5 and M6. The last three are mainly used for judging whether a loop is a kernel or not. MLL represents maximum loop nest level where a loop is placed in the call tree, which reflects the order of the number of potential iterations. MFL and MMA are from the perspective of specific code optimization patterns: loop fusion and array merging. Both patterns in a loop are easily identified syntactically and can be an indication of a kernel.

Loop fusion is a loop transformation which combines two adjacent loops that have the same iteration range into a single loop. It can improve temporal and spatial localities of data references. Array merging is a data layout optimization technique that merges two or more arrays of the same size into a single array. It can improve the spatial locality between elements of different arrays.

3.2 Estimation Schemes for $\mathsf{B/F}$

As mentioned above, $\mathsf{B/F}$ plays a primary role in the kernel class identification. Actual $\mathsf{B/F}$ of a specific code region is usually measured dynamically by profiling or tracing. Although we cannot statically obtain cache miss rate and hence actual $\mathsf{B/F}$ in general, we statically estimate $\mathsf{B/F}$ based on heuristics that are derived from the experience of performance tuning.

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 simply count the unique signatures of array references found in a loop. We provide three estimation schemes, ES0–2, each of which has its own method of calculating the signature of an array reference. All of the schemes assume that the *abstract syntax tree (AST)* of an array reference is available.

ES0 For the signature of an array reference $a(x)$, the scheme calculates hash value of the AST of $a(x)$.

ES1 For the signature of an array reference $a(x_1...)$, the scheme calculates hash value of the AST of $a(\bullet...)$ derived from the original by anonymizing the first dimension.

ES2 In addition to ES1, for the signature of an array reference $a(x_1, \mathsf{op}(x_2, c)...)$, the scheme calculates hash value of the AST of $a(\bullet, x_2...)$ derived from the original by modifying it in a way that the addition or subtraction (op) of constant c are ignored at the second dimension.

ES0 only assumes that data is shared in cache 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 by additions/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.

```
1  integer m
2  integer n1,n2,n3
3  double precision u(n1,n2,n3),v(n1,n2,n3),r(n1,n2,n3),a(0:3)
4  integer i3,i2,i1
5  double precision u1(m), u2(m)
6  do i3=2,n3-1
7     do i2=2,n2-1
8        do i1=1,n1
9           u1(i1) = u(i1,i2-1,i3) + u(i1,i2+1,i3)&
10                 + u(i1,i2,i3-1) + u(i1,i2,i3+1)
11          u2(i1) = u(i1,i2-1,i3-1) + u(i1,i2+1,i3-1)&
12                 + u(i1,i2-1,i3+1) + u(i1,i2+1,i3+1)
13       enddo
14       do i1=2,n1-1
15          r(i1,i2,i3) = v(i1,i2,i3)&
16                      - a(0) * u(i1,i2,i3)&
17                      - a(2) * ( u2(i1) + u1(i1-1) + u1(i1+1) )&
18                      - a(3) * ( u2(i1-1) + u2(i1+1) )
19       enddo
20    enddo
21 enddo
```

Listing 1: A loop kernel

Suppose that we are estimating B/F's for a loop kernel shown in Listing 1 taken from MG (Multi-Grid), which is a component of NPB[1]. The kernel has 15 floating-point operations. The volume of memory accesses is different according to estimation scheme. By ES0, we count all occurrences of the array references, that is, 3 stores and $18 + 3$ loads (in the case of SPARC64VIIIfx, also a load is issued for an array reference at the left hand side of an assignment). Thus, the B/F (ES0) is estimated to be $8 * 21/15 = 11.2$. We count by ES1 a store at line 15 as well as 2, 2, 2, 2, 2, and 1 loads at lines 9, 10, 11, 12, 15, and 16, respectively. Thus, the B/F (ES1) is estimated to be $8 * 12/15 = 6.4$. By ES2, we count a store at line 15 as well as 1, 2, and 2 loads at lines 9, 10, and 15, respectively. Thus, B/F (ES2) is estimated to be $8 * 6/15 = 3.2$.

[1]http://www.nas.nasa.gov/publications/npb.html

4. ANALYZING THOUSANDS OF FORTRAN APPLICATIONS

For the purpose of investigating the distribution of kernel classes across thousands of computation-intensive applications, we have developed a tool that is composed of a Fortran parser and a *factbase*. A factbase is filled with facts about the ASTs and the semantic information extracted by the parser. Source code features are obtained by querying the factbase.

4.1 Parsing for Source Code Survey

Extracting features of loop kernels from source code requires an understanding of the code's semantics instead of simple string pattern matching. The most basic requirement is to *parse* it [5] to obtain its AST. If we employ ordinary compiler frontends for parsing source code, we have to *build* applications. In order to parse tens of million lines of source code from thousands of Fortran applications in reasonable time, we would have to overcome the following obstacles that hinder the development of automated analysis.

Building applications By searching the root directories of 2000 repositories tagged with "language:FORTRAN" hosted on GitHub, we found that approximately 800 of the root directories contain `Makefile` or its variation which suggests the use of "make" command. We also found that other 100 root directories contain `CMakeLists.txt` which suggests the use of "cmake" command, while we could not identify how to build for the rest of the repositories.

Specifying configuration options By observing several of the repositories in more detail, we have noticed that quite a few of the applications use preprocessor directives such as `#ifdef` for adapting to different compilation/computing environments. For example, WRF (the Weather Research & Forecasting model)[2] has hundreds of `#ifdef` variables. Obviously, it is not practical to handle the whole possible combinations of such variables.

Covering language variants Since Fortran is one of the longest-lived programming languages, there exist several standards incompatible with each other. Moreover, some applications depend on vendor-specific language extensions or dialects, which would require us to purchase a number of commercial compilers.

Preparing external libraries Without the help of a package management system, we have to manually install the required external libraries.

To cope with the situation, we employ a dedicated Fortran parser that we have developed for the previous work [14]. The parser is based on the following major standards: FORTRAN77, Fortran90, Fortran95, Fortran2003, and Fortran2008. It is also capable of handling the following.

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

[2]http://www.wrf-model.org/

Directives It can also 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 since we made 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 also 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. For example, since both an array access and a function reference are written in the same form like a(x), the type of a is required to disambiguate a(x).

The parser has been intensively tested for numerous applications. 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 could disambiguate them later by resolving dangling references in a factbase.

4.2 Source Code Facts and Ontologies

As mentioned in Section 4.1, our parser requires deferred resolution of dangling references or ambiguous symbols. Moreover, a loop classification task requires call trees to obtain maximum loop nest level (MLL) as seen in Section 3. Thus, we need a database for storing facts extracted from multiple source files by the parser. Feature extraction is performed later by querying the database.

As the database queries may contain syntactic AST patterns and/or call graph patterns, it is natural to use tree/graph databases rather than conventional relational databases. We use RDF (Resource Description Framework)[6] store for the databases. 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. We do not have to describe all kinds of statements when referring to "statement" in a query, for example.

A fact about AST and semantic information extracted by the parser is described as a triple of *subject*, *predicate* (also called *property*), and *object* following RDF. Both subject and object may be source code entities such as files, functions/subroutines, and statements. A predicate denotes a binary relation between a pair 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`. Conceptual classes of entities and predicates such as "name" above are specified by ontologies. Ontologies define concepts and relationships used for describing facts. We have designed the following ontologies in the OWL ontology language[7].

[3]http://openmp.org/
[4]http://openacc.org/
[5]http://cristal.inria.fr/~fpottier/menhir/
[6]http://www.w3.org/RDF/

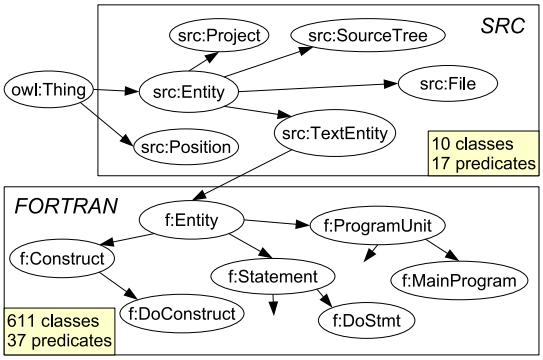


Figure 2: Classes in ontologies (excerpts)

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

FORTRAN An ontology for Fortran languages that defines the classes of Fortran entities based on specifications such as FORTRAN77, Fortran90, and other dialects.

In OWL, a class is defined as a subclass of owl:Thing. We disambiguate the names of conceptual classes by prefixing namespaces to the names like owl:Thing or src:Entity. We use namespaces listed in the following table throughout the rest of the paper.

Prefix	Meaning
xsd:	XML Schema Datatypes
rdf:	Resource Description Framework
rdfs:	RDF Schema
owl:	OWL Web Ontology Language
src:	Core source code entity
f:	Fortran source code entity

Figure 2 illustrates the hierarchy of conceptual classes in the ontologies. *SRC* defines classes including src:TextEntity for source code entities represented as texts. *FORTRAN* defines classes for Fortran entities as subclasses of f:Entity, which in turn is a subclass of src:TextEntity.

In addition to the classes above, we also have defined predicates for each ontology. There exist two types of predicates in OWL: *object property*, which is a relation between instances of conceptual classes, and *datatype property*, which is a relation between instances and RDF literals or possibly values of XML schema datatypes[8]. A predicate is defined as a subproperty of owl:ObjectProperty or owl:DatatypeProperty.

Excerpts of the predicates in *SRC* are listed in Table 2. The first five predicates are object properties and the rest is a datatype property. A predicate src:inFile is a subproperty of src:containedIn. The domain and the range of a predicate are also specified. Note that the predicates in *SRC* include

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

Predicate	Domain	Range
src:parent	src:Entity	src:Entity
src:children	src:Entity	rdf:List
src:containedIn	src:Entity	src:Entity
⊢ src:inFile	src:Entity	src:File
src:startPosition	src:Entity	src:Position
src:line	src:Position	xsd:nonNegativeInteger

Table 2: Predicates in source code ontology

Predicate	Domain	Range
f:inProgramUnit	f:Entity	f:ProgramUnit
⊢ f:inModule	f:Entity	f:Module
f:loopControl	f:DoConstruct	f:LoopControl
f:name	f:Entity	rdfs:Literal

Table 3: Predicates in Fortran ontology (excerpts)

src:parent and src:children. They define parent-children relationships in terms of hierarchical structures such as directory trees and ASTs. Table 3 gives excerpts from the predicates defined in *FORTRAN*. The first three predicates are object properties and the last one is a datatype property. A predicate f:inModule is a subproperty of f:inProgramUnit. The domain and the range of a predicate are also specified. Since f:Module is a subclass of f:ProgramUnit, a fact $(e, \text{f:inModule}, p)$ entails $(e, \text{f:inProgramUnit}, p)$.

4.3 Factbase Query for Feature Extraction

In order to search factbases for loop features, we write queries for the patterns in SPARQL[9]. SPARQL is a standard query language for RDF. Roughly speaking, SPARQL is an extension of SQL with graph patterns described by a set of triples with *query variables*. For example, consider the following query that will enumerate names of all main programs in the factbase, where identifiers prefixed by "?" denote query variables.

```
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. A graph pattern is essentially a set of triples written in the format

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

that may contain abbreviation symbol "a" for rdf:type that is used to specify a conceptual class of an entity. Consecutive triples that share a subject can also be written as

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

Note that predicates and ontology classes are prefixed. Prefix bindings that are placed in the head of a query are omitted for brevity.

[9]http://www.w3.org/TR/sparql11-query/

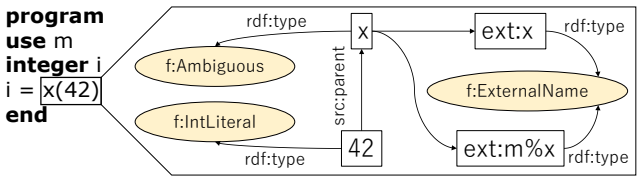


Figure 3: Fact graph for ambiguous reference x(42)

4.3.1 Deferred Reference Resolution

As mentioned in Section 4.1, our parser may yield ambiguous AST nodes due to dangling references. The dangling references are however easily resolved by querying the factbase augmented by inserting f:refersTo facts. This is achieved by the following SPARQL Update[10] snippet

```
INSERT {
 ?ent f:refersTo ?provider .
}
WHERE {
 ?provider f:provides ?ext .
 ?ent f:requires ?ext .
}
```

that adds f:refersTo facts that link f:provides facts to f:requires facts, which are also generated by the parser. Suppose that we have a program shown in Figure 3 that contains ambiguous reference x(42). According to the RDF data model, a set of facts form a directed graph, where each triple (s, p, o) is represented by a graph fragment $s \xrightarrow{p} o$. By parsing the fragment, we obtain a set of facts that corresponds to the fact graph illustrated in Figure 3, where ext:x and ext:m%x denote possible external names that may be required by x. Then the ambiguous x(42) will be resolved to an array element if module m provides array x, or to a function reference if m provides function x. Otherwise, x may be defined elsewhere at the top level. If both of the external names are defined, ext:m%x should precede based on the scoping rule.

4.3.2 Feature Extraction

The loop features explained in Section 3 are obtained by querying the factbase. As an example, we consider the number of floating-point operations (FOp). The query shown in Listing 2 counts distinct (sub-)expressions in a loop relying on hash values of the sub-ASTs.

OPTIONAL clause at lines 5 through 26 specifies optional parts of the graph pattern. In the optional match, either the optional graph pattern matches a graph, thereby variables ?loop and ?nfop in the pattern are bound to solutions, or the variables are left null. The subquery at lines 6 through 25 calculate aggregate value COUNT(DISTINCT ?h) for each ?loop specified by **GROUP BY**. Each **FILTER** clause limits solutions to the ones satisfying the condition: logical and concatenation operators are filtered out at line 13, and the solutions are further limited at lines 18 through 24 to the ones that the operands are *real-literal-constant* or expressions containing variables of floating-point type such as REAL. A SPARQL property path src:parent+ is used at line 16 for specifying a path of one or more occurrences of src:parent.

[10]https://www.w3.org/TR/sparql11-update/

```
SELECT DISTINCT ?loop ?nfop WHERE {
 ?loop a f:DoConstruct ;
       f:inProgramUnit ?pu .

 OPTIONAL {
  SELECT ?loop (COUNT(DISTINCT ?h) AS ?nfop)
  WHERE {
   ?fop a f: IntrinsicOperator  ;
        src:treeDigest  ?h ;
        a ?fop_cat  ;
        f:inDoConstruct ?loop .

   FILTER (?fop_cat NOT IN (f:Not, f:And, f:Or,  f:Concat))

   ?opr a f:Expr ;
        src:parent+ ?fop .

   FILTER (EXISTS { ?opr a f:RealLiteralConstant } ||
           EXISTS {
             ?opr f: declarator  ?dtor  .
             ?dtor a f:Declarator  ;
                   f:declarationTypeSpec ?tspec  .
             ?tspec a f:FloatingPointType  .
           })
  } GROUP BY ?loop
 }
}
```

Listing 2: Query for counting floating-point ops

Feat.	Min.	25%	Med.	75%	Max.	Mean ± S.D.
AR0	1	5	8	16	1199	15.49 ± 30.01
AR1	1	2	4	7	739	6.33 ± 12.15
AR2	1	2	3	6	739	6.04 ± 11.72
FOp	1	2	6	16	5326	21.87 ± 65.26
St	2	5	9	20	2820	22.67 ± 52.72
Br	0	0	0	2	794	2.30 ± 7.96
Ca	0	0	1	4	1550	4.32 ± 14.64
MAR	1	2	2	2	7	2.29 ± 0.74
MLD	1	1	2	3	18	2.12 ± 1.19
IAR0	0	0	0	0	463	0.45 ± 4.05
IAR1	0	0	0	0	150	0.22 ± 2.11
IAR2	0	0	0	0	232	0.21 ± 1.80
MLL	0	0	2	4	29	2.92 ± 3.26
MFL	0	0	1	1	109	1.06 ± 1.86
MMA	1	2	2	4	162	3.71 ± 4.59
BF0	0.05	5.65	11.64	24.00	2856.0	18.92 ± 29.24
BF1	0.01	2.00	5.33	12.00	1248.0	9.60 ± 15.04
BF2	0.01	2.00	4.80	12.00	1248.0	9.34 ± 14.67

Table 4: Statistics of loop features (quartiles, mean, and standard deviation)

5. EXPERIMENTS

In this section, we report on our experiment conducted on loops in a thousand Fortran applications. All experiments described in the rest of this section were performed on a workstation with an 8-core Intel Xeon processor (3.0 GHz) with 64GB RAM.

5.1 Surveying Loops Across Applications

5.1.1 Analyzing Fortran Applications on GitHub

We have cloned approximately 2000 popular repositories tagged with "language:FORTRAN" hosted on GitHub. We evaluated the popularity of repositories based on the number of "stars" they have. Among the cloned repositories, 1180 were selected based on the occurrence of keywords such as "simulation" or those indicating the use of OpenMP, OpenACC, MPI to filter out irrelevant applications. Then by parsing the source code contained in the repositories, we extracted facts about the loops that are reachable from main programs and then finally selected 175 963 loops from 1020 repositories based on the occurrences of non-trivial arrays and floating-point operations. More precisely, we selected the loops that have non-zero B/F by ES2. It took approximately 10 hours to extract the source code facts from 1,180 repositories (58M source lines of code, 220K files), 24 hours to create a factbase of the facts, and 37 hours to extract the loop features.

Table 4 gives the statistics of the loop features: the quartiles, the means, and the standard deviations, where the suffixes 0, 1, and 2 for AR, IAR, and BF indicate the estimation schemes ES0, ES1, and ES2, respectively. The histograms of the features over the 175,963 loops are shown in Figure 4. The values of St, AR0, FOp, IAR0, Br, and Ca are shown up to 100 to magnify the distribution. Nevertheless, they still cover more than 95% of the samples. Similarly, those of BF0, BF1, and BF2 are shown up to 60.0. We omit AR1, AR2, IAR1, and IAR2 since they are quite similar to AR0 or IAR0.

The distributions look like skew-normal distributions except those of BF0, BF1, and BF2 in that they have strange spikes at regular intervals of 4 or 8. We can observe similar spikes at a cycle of 1.0 in the distribution of the quotients of two positive integers as shown in Figure 5. Note that B/F ratio is the quotient of integers, that is, the number of transferred bytes and the number of floating-point operations. Since the number of the bytes depends on the type of transferred data, the cycle is multiplied by 4 or 8, which are the sizes for REAL or DOUBLE PRECISION, respectively.

5.1.2 Manual Classification of Sampled Loops

From the set of loops, 100 loops were randomly sampled and then manually classified into the six classes by Terai (junior performance engineer, 5 years of experience) and Minami (expert performance engineer, 30 years of experience). In addition to knowledge of computer architectures, compilers, programming languages and numerical algorithms, the task requires the ability to understand what kind of computation (e.g. stencil code) a given loop implements by inspecting the source code and by consulting reference materials if needed.

In order to support classifying the loops, we developed a web application that shows them the sampled loops and receives the classification results. A snapshot of the web application is shown in Figure 6, where a sampled loop decorated with loop features such as estimated B/F is highlighted over the outline of the AST and the call tree of the entire application. The experts performed the classification task based on the criteria explained in Section 2.1 consulting the feature values and the source code. The results were submitted to another database through the class selection menu.

5.1.3 Results

Table 5 and Figure 7 show the distribution of kernel classes and 95% confidence intervals (in parentheses and by error bars, respectively) for each expert. They indicate that M5 kernels are in the majority, accounting for approximately 60% (95% confidence interval: 50%–70%) of the sampled *kernels*. It should be also noted that "OtherKernel" is second to M5 and accounts for a relatively large fraction. Some

	Terai	Minami
Nonkernel	0.36 (0.27–0.45)	0.23 (0.15–0.31)
M1	0.05 (0.01–0.09)	0.05 (0.01–0.09)
M2	0.01 (0.0–0.03)	0.09 (0.03–0.15)
M3	0.01 (0.0–0.03)	0.00 (n/a)
M4	0.03 (0.0–0.06)	0.02 (0.0–0.05)
M5	0.39 (0.29–0.49)	0.44 (0.34–0.54)
M6	0.03 (0.0–0.06)	0.03 (0.0–0.06)
OtherKernel	0.12 (0.06–0.18)	0.14 (0.07–0.21)

Table 5: Classification results

kernels were judged to be OtherKernel due to the lack of information such as source code of external libraries, while others since they are (part of) test programs.

As mentioned in Section 3, our classes of computational kernels are closely related to B/F, that is, M1–4 kernels require low $\mathcal{B}/\mathcal{F}$, while M5–6 kernels high. Figure 8 shows how average B/F's are distributed over classes identified by the junior expert, Terai (left) and the senior expert, Minami (right). We can see that average B/F's for M5 and M6 are higher than those of M3 and M4. It should be noted that average B/F's for M1 and M2 are relatively high, though both classes require low $\mathcal{B}/\mathcal{F}$ by definition. This is because both experts inspected the loops qualitatively regardless of the B/F relying on the comment lines or their expertise in numerical algorithms.

5.2 Predicting Loop Kernels

This section explains how we can predict loop kernels for a given application by means of the classification results and statistical classification techniques. We rely on *supervised learning* to classify a loop according to whether it is a kernel or not. Supervised learning is a machine learning technique that requires a set of examples called *training set* to construct predictive models. In this case, an example is composed of a set of loop features and a kernel class. We employ the Minami's classification results for the training set. We represent a set of source code features as an n-dimensional vector of numerical values, called *feature vector*. This means that we characterize a code fragment simply by n attributes.

A training set is a set of examples

$$\{(\mathbf{x}, c)|\mathbf{x} \in \mathbb{R}^n, c \in \{\text{NonKernel}, \text{Kernel}\}\},$$

where $\mathbf{x}$ and c are a feature vector and a binomial class, respectively. For example, $(\mathbf{x}, \text{NonKernel})$ means a loop characterized by $\mathbf{x}$ is not a kernel. By applying a classification algorithm we obtain a classification function $p \in \mathcal{P} : \mathbb{R}^n \to \{\text{NonKernel}, \text{Kernel}\}$ that predicts c from $\mathbf{x}$.

From the features mentioned in Section 3, we selected the following seven attributes by analyzing their correlation with the class (kernel or not) in order to maximize the classification accuracy: MLL, MLD, MMA, MFL, IAR0, Ca, and BF0, where IAR0 and BF0 denote the number of indirect array references and B/F both of which are calculated by ES0, respectively.

We employed C-SVC (C-Support Vector Classification) in LIBSVM [6] from a data mining library called scikit-learn[11] to construct a predictive model from the training set. The parameters were chosen following a cookbook [15], that is, $C = 32$, kernel=RBF (Radial Basis Function), and $\gamma = 8$. It took 0.4 seconds to construct the model.

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

Classification Accuracy: 81.00% (100 examples)

	True Kernel	True NonKernel	Prec.
Pred. Kernel	73	16	82.02%
Pred. NonKernel	3	8	72.73%
Recall	96.05%	33.33%	

Table 6: Evaluation of loop kernel prediction

5.2.1 Evaluation of Loop Kernel Prediction Model

The result of 20-fold cross validation of the model is summarized in Table 6, where precision (abbreviated by prec.) means the proportion of true positives among instances classified as positive, recall means the proportion of true positives among all positive instances, and classification accuracy means the proportion of correctly classified examples.

Except for disappointing recall of non-kernel prediction, all scores look promising. The result indicates the correlation between static features of a loop and the likelihood that it is a kernel, although it does not immediately entail that the classifier predicts kernels with 80% accuracy for unknown applications.

5.2.2 Predicting Kernel Classes

Although our training set does not have sufficient examples for each kernel class other than M5, we made an attempt to construct a model for predicting the class of a given kernel. We employed again LIBSVM as it supports *multi-class classification*. A training set is a set of examples $\{(\mathbf{x}, c)|\mathbf{x} \in \mathbb{R}^n, c \in \{\text{M1}, \text{M2}, \text{M4}, \text{M5}, \text{M6}, \text{OtherKernel}\}\}$, where $\mathbf{x}$ and c are a feature vector and a polynomial class, respectively. Note that M3 is omitted since the training set is taken from the Minami's classification result and also that the training set is shrunk by filtering out the examples each of which were classified as non-kernel, hence the size becomes 76. For constructing a classifier, we selected three attributes St, Br, and Ca in the same way as the binomial classifier above. The result of leave-one-out cross validation of a model by means of C-SVC with $C = 16$, kernel=RBF, and $\gamma = 0$ is summarized in Table 7. We adopted leave-one-out since the number of examples are significantly decreased. It took less than 1 second to create and to evaluate the model.

As expected, we could not construct a classifier for distinguishing kernel classes since the training set does not have sufficient examples of kernel classes except M5. Nevertheless, the result suggests that a classifier for distinguishing M5 kernels from the others would be promising and again indicates the correlation between static features and the likelihood of being an M5 kernel.

6. RELATED WORK

There is a large body of empirical studies that surveyed a considerable amount of source code [17, 7, 12, 13, 18, 10, 9, 19]. We discuss only a few of them due to space limitations.

An empirical study conducted by Knuth [17] is one of the earliest studies that analyze a number of Fortran programs. He and his collaborators performed source code analysis on over 440 programs consist of 250 000 punched cards, each of which corresponds to a line of code. They counted the statements in the programs for each statement type (e.g., assignment, IF, DO, etc.), calculated the length and the depth of nesting of 7933 DO-loops, rated the complexity of 83 304 assignment statements by counting operators ('+'

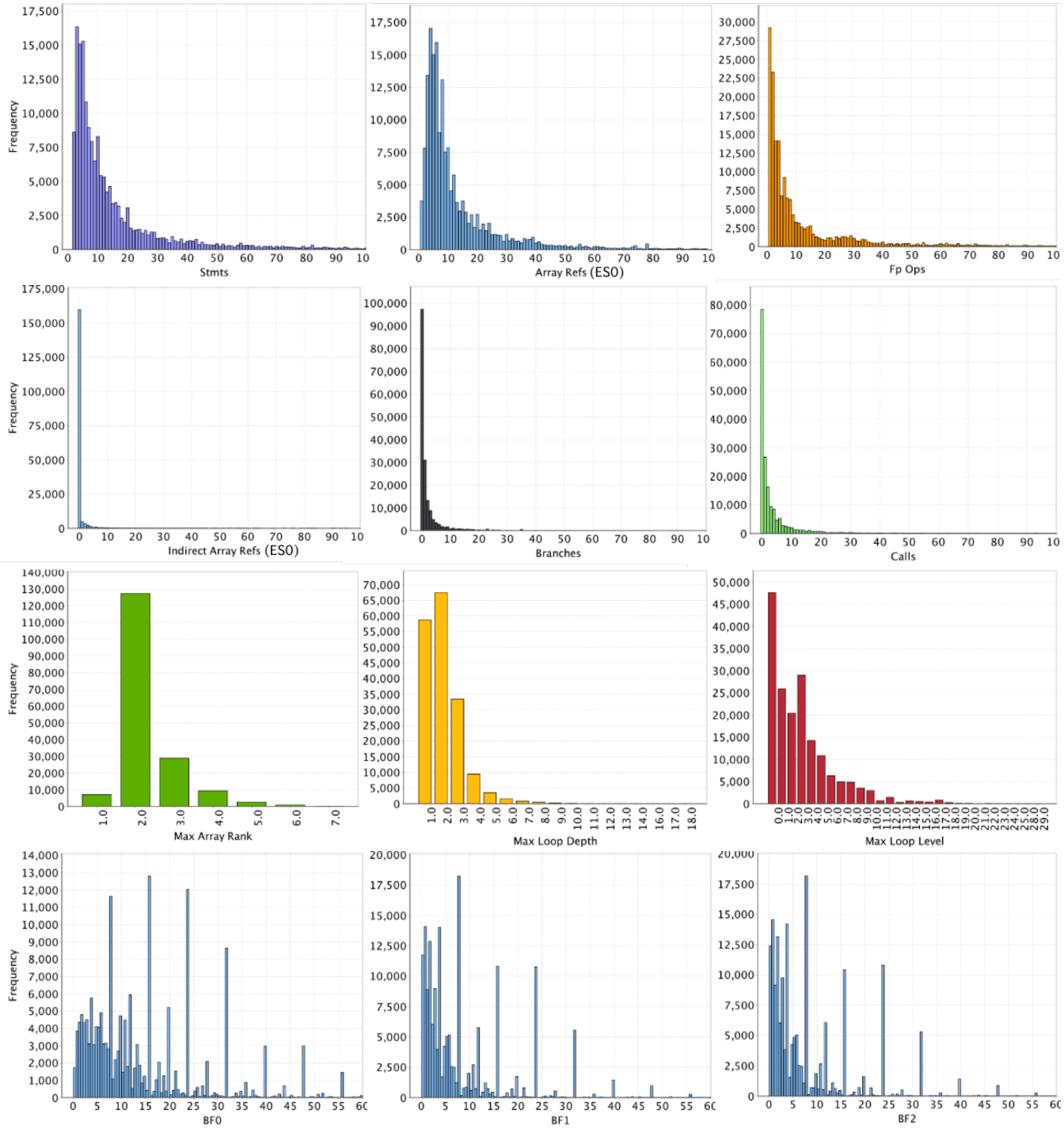


Figure 4: Distribution of loop features

Classification Accuracy: 65.79% (76 examples)

	True M1	True M2	True M4	True M5	True M6	True OtherKernel	Prec.
Pred. M1	2	0	0	0	0	0	100.00%
Pred. M2	0	1	0	0	0	1	50.00%
Pred. M4	0	0	0	0	0	0	0.00%
Pred. M5	3	6	0	40	2	5	71.43%
Pred. M6	0	0	0	0	0	0	0.00%
Pred. OtherKernel	0	2	2	4	1	7	43.75%
Recall	40.00%	11.11%	0.00%	90.91%	0.00%	53.85%	

Table 7: Evaluation of kernel class prediction

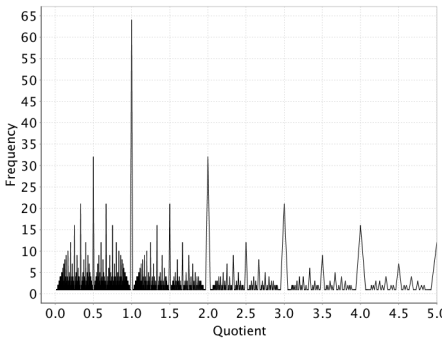


Figure 5: Distribution of quotients (positive integers up to 64)

and '−' were counted as 1, '∗' was counted as 5, etc.), and so forth. In addition, they performed run-time analysis on 25 programs randomly selected out of the target programs. To find hotspots in a program, they counted how many times each statement in a program was executed by inserting instrumentation code into the program. Moreover, based on the run-time analysis, they identified kernels of randomly selected 17 programs.

By virtue of technological advances in the past 45 years, we can analyze 58 million source lines of code from 1020 repositories compared to 250 000 lines from 440 programs. In addition, we performed elaborated B/F calculations and conducted experiments in predicting and classifying loop kernels with a machine learning approach, while their work simply calculated the number of statements, the length and depth of loops, the complexity of assignment statements. We did not, however, conduct run-time analysis in this study unlike theirs. Our previous work [14] made use of performance profiling though the number of analyzed projects was very limited like theirs.

Dyer and others conducted a large-scale empirical study of 31,432 Java projects [10]. The reason why they could study much more projects than ours is that the number of Fortran projects found in public repositories is much smaller than that of Java projects (they analyzed Java projects hosted on SourceForge, while we used Fortran projects hosted in GitHub). They focused on analyzing the usage of Java language features, while we focused on analyzing characteristics of loop kernels of Fortran programs. Although the goal of their work differs from ours, there is one common point; both works implemented infrastructures for mining source code repositories.

With the increase of the size of the target and of the complexity of the properties to be analyzed, it becomes harder to write analysis programs that directly accesses the target source code repositories. Dyer and others designed and implemented a domain-specific language (DSL) and its infrastructure called Boa [9] for mining source code. With Boa, users can write analysis programs concisely and efficiently because it automatically obtains data from source code repositories and prepares data structures that are necessary for simple description of mining tasks. In addition, Boa is capable of distributing specified mining tasks to a cluster system automatically, thus users can enjoy the benefit of the cluster without worrying about the details of task distribution.

On the other hand, we implemented a factbase, which is a database of facts about ASTs and the semantic information obtained by parsing Fortran programs. By writing and running queries on the factbase, users are able to analyze simple to complex properties of the source code repositories stored in the factbase. Regarding distributed processing of queries, since the database software we utilized in this study (Virtuoso) is capable of running on a distributed cluster system, users should be able to enjoy the benefit of the cluster for free, though we have not tested distributed processing on a cluster system yet.

Both of DSL approach and factbase approach have their own pros and cons. For example, our factbase approach should be able to handle more complex properties with simpler queries than their DSL approach because our approach utilizes RDF database with the elaborated query language SPARQL and rich vocabularies, while their approach utilizes simple data structures derived from ASTs by using the visitor-pattern. On the other hand, their approach would be able to process analysis tasks more efficiently than our approach because Boa employs map-reduce style data processing and its infrastructure utilized Hadoop, which is a highly scalable and efficient distributed data storage and processing framework. Fortunately, both approaches are orthogonal to each other, hence it should be feasible and promising to integrate them.

7. LIMITATIONS

In this study, all repositories are taken only from GitHub. Of course, we cannot assume that the Fortran applications hosted on GitHub represent the computation-intensive applications all over the world. Moreover, there are several potential perils such as "*A repository is not necessarily a project*" in mining repositories on GitHub [16]. Nevertheless, we have avoided those by carefully filtering the repositories as explained in Section 5.1. Note that it is also possible that some computation-intensive loops may be overlooked due to parsing errors or to the absence of the use of MPI, OpenMP, or OpenACC.

Due to lack of time, the kernel identification and the kernel class prediction functions have not been extensively evaluated on loops other than the sampled ones yet. They might have to be evaluated based on whether they can detect bottlenecks in an application, even though they were made relying only on manual classification results without measuring the runtime performance. In general, however, it is a very time-consuming task. Even preparing proper input data for the measurement is far from straightforward.

Another consideration is about test programs. Although 11 of the sampled 100 loops were parts of test programs, we did not exclude them since it is not trivial to identify test programs. Actually, 7 of the 11 loops were located in files whose path names did not contain "test" regardless of capitalization.

8. CONCLUSION

The process of performance tuning is time consuming and costly even if it is carried out automatically. It is crucial to share the experience in optimizing similar computational kernels. In this study, we explored a thousand computation-intensive applications written in Fortran to reveal the distribution of kernel classes, each of which is related to expected

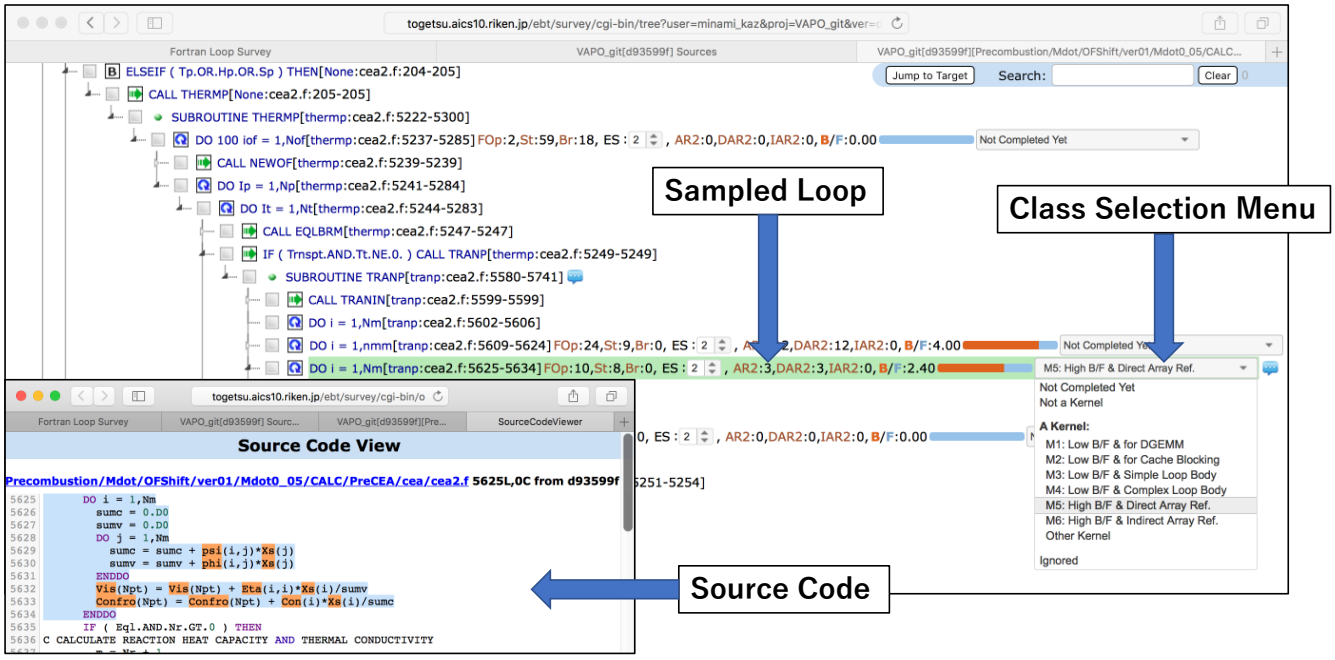


Figure 6: A web application supporting manual loop classification

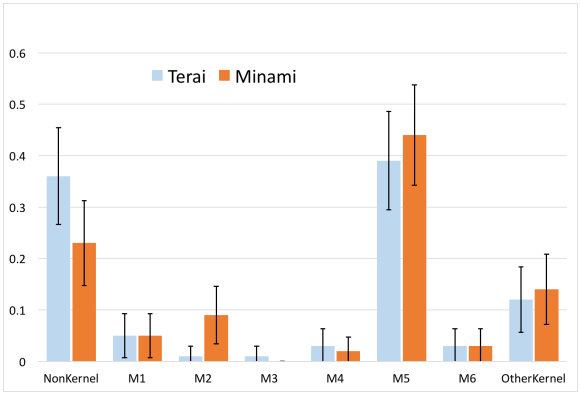


Figure 7: Classification results

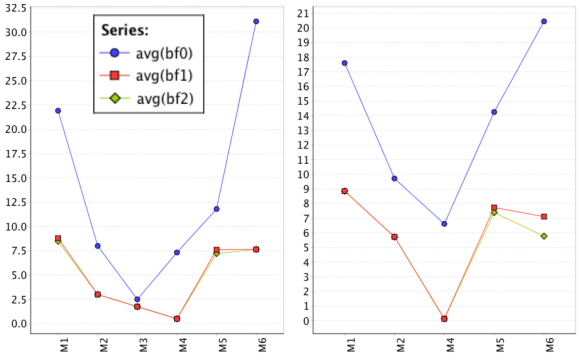


Figure 8: Average B/F over kernel classes (left:Terai, right:Minami)

efficiency and specific tuning patterns. Such exploration is a significant step toward the development of computational kernel identification as well as tuning pattern prediction to facilitate evidence-based performance tuning.

We constructed a factbase, or a database of source code facts obtained by parsing the whole source code contained in the applications. Then we extracted static features of the loops from the factbase and selected 175 963 of them based on the occurrence of non-trivial array references and floating-point operations. To estimate the distribution of the kernel classes, 100 loops were randomly sampled and then manually classified by experienced performance engineers[12].

The result indicates that 15–45% of the 175 963 loops are non-kernel loops and that 30–50% belong to a class of memory-bound kernels such as standard stencil computations, which require a high ratio of memory accesses to floating-point operations and hence are difficult to execute efficiently on modern scalar processors.

In addition to the loop classification, we discussed the correlation between the static features and the likelihood of belonging to a kernel class by constructing a binary classifier for identifying loop kernels and a multi-class classifier for predicting kernel classes, based on the manual classification data and a machine learning algorithm. The results of cross-validation of the classifiers indicate the correlation between static features of a loop and the likelihood of being a kernel, although the experts inspected the loops qualitatively depending on the information including comment lines as well as the static features.

Last but not least, the framework we employed for the experiment is not limited to a single programming language, a single classification scheme, nor a single repository hosting

[12]The results of the experimentation with supplementary information are publicly available at https://github.com/ebt-hpc/icpe2017

service. It would be possible to analyze applications written in more popular languages such as Java and C, since we have parsers for them and can make use of external static analyzers to extract call graphs. It would also be interesting to roughly estimate the cost of a given loop based on the profiling data of randomly sampled loops.

Acknowledgments

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

9. REFERENCES

[1] K. Asanović, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.

[2] D. H. Bailey, R. F. Lucas, and S. W. Williams. *Performance Tuning of Scientific Applications*. CRC Press, 2011.

[3] 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*, 25(4):29–36, 2008.

[4] 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*, 27(4):379–393, 2013.

[5] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler. A few billion lines of code later: Using static analysis to find bugs in the real world. *Commun. ACM*, 53(2):66–75, 2010.

[6] C.-C. Chang and C.-J. Lin. Libsvm: A library for support vector machines. *ACM Trans. Intell. Syst. Technol.*, 2(3):27:1–27:27, 2011.

[7] C. Collberg, G. Myles, and M. Stepp. An empirical study of java bytecode programs. *Softw. Pract. Exper.*, 37(6):581–641, 2007.

[8] J. Dongarra and P. Luszczek. HPC challenge benchmark. http://icl.cs.utk.edu/hpcc/.

[9] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen. Boa: Ultra-large-scale software repository and source-code mining. *ACM Trans. Softw. Eng. Methodol.*, 25(1):7:1–7:34, 2015.

[10] R. Dyer, H. Rajan, H. A. Nguyen, and T. N. Nguyen. Mining billions of ast nodes to study actual and potential usage of Java language features. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, pages 779–790, 2014.

[11] 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*, 39(3):296–327, 2011.

[12] T. Gorschek, E. Tempero, and L. Angelis. A large-scale empirical study of practitioners' use of object-oriented concepts. In *Proceedings of the 2010 ACM/IEEE 32nd International Conference on Software Engineering (ICSE)*, volume 1, pages 115–124, 2010.

[13] M. Grechanik, C. McMillan, L. DeFerrari, M. Comi, S. Crespi, D. Poshyvanyk, C. Fu, Q. Xie, and C. Ghezzi. An empirical investigation into a large-scale Java open source code repository. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 11:1–11:10, 2010.

[14] M. Hashimoto, M. Terai, T. Maeda, and K. Minami. Extracting facts from performance tuning history of scientific applications for predicting effective optimization patterns. In *Proceedings of the 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories (MSR)*, pages 13–23, 2015.

[15] C.-W. Hsu, C.-C. Chang, and C.-J. Lin. A practical guide to support vector classification. Technical report, National Taiwan University, 2003. http://www.csie.ntu.edu.tw/~cjlin/papers/guide/guide.pdf.

[16] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian. The promises and perils of mining github. In *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR)*, pages 92–101, 2014.

[17] D. E. Knuth. An empirical study of FORTRAN programs. *Software: Practice and Experience*, 1(2):105–133, 1971.

[18] R. Lämmel, E. Pek, and J. Starek. Large-scale, AST-based API-usage analysis of open-source Java projects. In *Proceedings of the 2011 ACM Symposium on Applied Computing (SAC)*, pages 1317–1324, 2011.

[19] G. Pinto, W. Torres, B. Fernandes, F. Castor, and R. S. Barros. A large-scale study on the usage of Java's concurrent programming constructs. *J. Syst. Softw.*, 106(C):59–81, 2015.

[20] M. Terai, H. Murai, K. Minami, M. Yokokawa, and E. Tomiyama. K-scope: A Java-based Fortran source code analyzer with graphical user interface for performance improvement. In *Proceedings of the 41st International Conference on Parallel Processing Workshops (ICPPW)*, pages 434–443, 2012.

[21] A. Tiwari, J. K. Hollingsworth, C. Chen, M. Hall, C. Liao, D. J. Quinlan, and J. Chame. Auto-tuning full applications: A case study. *Int. J. High Perform. Comput. Appl.*, 25(3):286–294, 2011.

[22] S. Williams, A. Waterman, and D. Patterson. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, 2009.

[23] 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)*, pages 371–372, 2011.