International Journal of Networking and Computing – www.ijnc.org
ISSN 2185-2839 (print) ISSN 2185-2847 (online)
Volume 5, Number 1, pages 180–199, January 2015

Identification and Elimination of Platform-Specific Code Smells in High Performance Computing Applications

Chunyan Wang
Graduate School of Information Sciences, Tohoku University
6-6-01, Aramaki-Aza-Aoba, Aoba-ku, Sendai, 980-8579, Japan

Shoichi Hirasawa
Graduate School of Information Sciences, Tohoku University
6-6-01, Aramaki-Aza-Aoba, Aoba-ku, Sendai, 980-8579, Japan

Hiroyuki Takizawa
Graduate School of Information Sciences, Tohoku University
6-6-01, Aramaki-Aza-Aoba, Aoba-ku, Sendai, 980-8579, Japan

and

Hiroaki Kobayashi
Cyberscience Center, Tohoku University
6-3, Aramaki-Aza-Aoba, Aoba-ku, Sendai, 980-8578, Japan

Received: August 1, 2014
Revised: November 3, 2014
Accepted: December 3, 2014
Communicated by Akihiro Fujiwara

Abstract

A code smell is a code pattern that might indicate a code or design problem, which makes the application code hard to evolve and maintain. Automatic detection of code smells has been studied to help users find which parts of their application codes should be refactored. However, code smells have not been defined in a formal manner. Moreover, existing detection tools are designed mainly for object-oriented applications, but rarely provided for high performance computing (HPC) applications. HPC applications are usually optimized for a particular platform to achieve a high performance, and hence have special code smells called platform-specific code smells (PSCSs). The purpose of this work is to develop a code smell alert system to help users find PSCSs of HPC applications to improve the performance portability across different platforms. This paper presents a PSCS alert system that is based on an abstract syntax tree (AST) and XML. Code patterns of PSCSs are defined in a formal way using the AST information represented in XML. XML Path Language (XPath) is used to describe those patterns. A database is built to store the transformation recipes written in XSLT files for eliminating detected PSCSs. The recall and precision evaluation results obtained by using real applications show that the proposed system can detect potential PSCSs accurately. The evaluation on performance portability of real applications demonstrates that eliminating PSCSs leads to significant performance

changes and therefore the code portions with detected PSCSs have to be refactored to improve the performance portability across multiple platforms.

Keywords: platform-specific code smell, XML representation, HPC

1 Introduction

In the field of high performance computing (HPC), the main concern is to fully utilize the potential of a specific target platform, which is determined by the hardware, the operating system, the compiler and the runtime libraries. In the case where an HPC application is optimized for a specific platform, it may become unable to run efficiently on other platforms. Namely, the performance is not portable; the performance portability of the application is low. When an application is ported to a new platform, hence, it is necessary to optimize the application again for the new platform. To lower such a software maintenance cost, therefore, an HPC application should be refactored so as to improve its performance portability.

The most common way to determine how to refactor an application code is to identify a code smell. A code smell is any part of an application code that potentially causes a design problem, which degrades the application maintainability. Common examples of code smells include duplicated code segments and long methods [1]. Refactoring tools to remove code smells have been developed to improve the readability, understandability and maintainability [2]. However, an HPC application is optimized for a particular platform, and the optimized code portions are sometimes considered code smells according to their conventional definitions. Hence, removing such code smells might degrade the performance. This paper refers to such optimized code portions as platform-specific code smells (PSCSs). A PSCS is a code pattern that is beneficial for some platforms but potentially harmful for other platforms.

Cong et al. [3] observed that performance analysis for identifying code smells requires in-depth knowledge of the application algorithm and the platform architecture. It remains challenging and time-consuming even for experienced users. It requires tremendous efforts to identify PSCSs and optimize an application for every target platform to achieve high performance portability. Therefore, there is a growing need for a supportive tool to detect PSCSs for evolving HPC applications.

Code smells have been studied in the research field of software engineering, and less attention is paid to PSCSs that are special code smells in HPC applications. Thus, studies on code smells have mainly discussed only software designs and architectures, and not considered platform-dependent aspects of an application. Although most of the conventional code smells are designed for object-oriented programming languages, many important HPC applications are written in C or Fortran. Furthermore, these code smells are mostly defined by textual description, which is informal. As a result, detection usually depends on manual inspection. It could be a tedious, time-consuming and error-prone task to find code smells with human intuition, especially when the application code is large. Therefore, we have to express the definitions of code smells in a formal way to identify PSCSs in HPC applications systematically.

The purpose of this work is to develop a code smell alert system for HPC applications to help users find PSCSs that potentially degrade performance portability across different platforms. The proposed PSCS alert system is based on an abstract syntax tree (AST) and Extensible Markup Language (XML) [4]. PSCS patterns are defined in a formal way using AST information represented in XML. XML Path Language (XPath) [5] is used to describe these patterns in a formal and standard way. A database is built to store the transformation recipes for eliminating detected PSCSs. Extensible Stylesheet Language Transformations (XSLT) [6] is used to write the transformation recipes. The recall and precision evaluation results demonstrate that the proposed system accurately detects PSCSs. The observation of compiler messages before and after eliminating PSCSs demonstrates that the detected PSCSs have a significant impact on utilization of the target platform. Performance evaluations of real applications that are originally optimized for a specific system are performed on different platforms. After eliminating PSCSs, these applications can get speedup on the target platforms. Therefore, these results demonstrate that the proposed system can accurately find code patterns, which are sensitive to the performance of a particular platform and hence should

be refactored for high performance portability.

The rest of this paper is organized as follows. Section II reviews the related work of code smell description and detection. Section III proposes a PSCS alert system, and discusses how to detect these PSCSs automatically. Section IV shows the evaluation results obtained using real applications. Finally, Section V gives concluding remarks.

2 Related Work

In this section, we review the related work of code smell description and the formal description for object-oriented applications, and then introduce code smell detection techniques and detection tools.

2.1 Code Smell Description

Several researches have been done on code smells. Fowler [1] identified 22 code smells and associated each of them with refactoring transformations that may improve the structure of code. Mantyla et al. [7] used a subjective taxonomy to categorize code smells identified in Fowler's book. Garcia et al. [8] described four representative architectural smells with textual descriptions. Ratzinger et al. [9] exploited historical data extracted from repositories, and pointed out that certain design fragments in software architectures can have a negative impact on system maintainability. They defined "change smell" as some parts of an application that are frequently modified at the same time. Then, they proposed an approach to detecting such change smells to improve the evolvability of an application.

However, since these code smells are defined informally, users have to manually identify where to refactor and which refactoring should be applied. It could cost tremendous efforts to find the code smells with human intuition especially when the application code is large. In the following subsection, we will introduce related researches about formal descriptions of code smells.

2.2 Formal Description for Object-Oriented Applications

Some researchers used XML to represent source codes and their intermediate objects, and showed that XML can be used as a portable source code representation since it is independent of programming languages. Putro et al. [10] proposed to represent program code forms as an XML format consisting of grammar, token stream (TS) and AST. Using information taken from a source code, TSs and ASTs, they proved the usability of this approach by using a code smell detector [11]. Slinger [12] used an AST-based detection to identify a number of code smells in Eclipse. Primitive code smells can be found directly in the source code by an analyser, and more complicated code smells can be derived by utilizing the smell aspect repository and Grok scripts [13]. As every compilation unit is fully parsed and analysed, AST represents the entire structure of the source code. Therefore, the design model generated by the AST-based approach is more precise.

However, conventional code smells are mainly described for object-oriented programming. The definitions of special code smells in HPC applications have not been established so far.

2.3 Detection Techniques

Travassos et al. [14] proposed an approach to the code smell detection based on manual inspection and reading techniques. However, manual inspection of the code to identify code smells based only on text-based description is a time-consuming and error-prone process. This detection technique does not scale to a large application code easily.

Longest common subsequence (LCS) is used in string matching when a program is represented as a text string [15]. The similarity-based heuristics string matching can detect the textual clones, but it is insufficient to detect all possible semantic code smells.

Some researches for detection of semantic clones used AST-based detection techniques. Yang developed an AST differencing algorithm [16]. Given a pair of two functions (f_T, f_R), the algorithm creates two ASTs, T and R, then uses the LCS algorithm, and matches their subtrees recursively.

This type of tree matching respects the parent-child relationship as well as the order of sibling nodes. As a result, it is sensitive to changes in nested blocks and control structures because tree roots must be matched at every level.

Hunt and Tichy [17] used syntactic information to guide string level differencing. Their 3-way merging tool parses a program into a language neutral form, compares token strings using the LCS algorithm, and finds syntactic changes using structural information from the parsing.

Neamtiu et al. [18] proposed an algorithm that tracks simple changes of variables, types, and functions based on an AST representation. Neamtiu's algorithm assumes that function names are relatively stable over time and match the ASTs of functions with the same name; the algorithm traverses two ASTs in parallel and incrementally adds one-to-one mappings as long as the ASTs have the identical structure. In contrast to Yang's algorithm, Neamtiu's algorithm cannot compare structurally different ASTs.

Every detection technique is an implementation of some pseudo equivalence functions. The more heuristics are used, the better the detection technique will work. Usually the detection techniques are based on the computation of different kinds of metrics. Fontana et al. [19] proposed a machine learning-based approach that computes a large set of metrics covering different aspects of the code to detect code smells.

All these detection techniques have contributed significantly to multi-version program analysis. However, they are either insufficient to detect all structural differences or sensitive to changes in nested blocks. Hence, none is sufficient to detect platform-specific code smells. Moreover, none of them provides a formal description of code smells.

2.4 Detection Tools

Some tools have been developed for the automatic detection of code smells. Most of the tools such as [20–22] can only analyze a specific language and do not take code structural information, which is related to the performance of an HPC application, into consideration.

Moha et al. [23] defined an approach that allows the specification and automatic detection of code and design smells. Their work is supported by a domain specific language for specifying four design smells using high-level abstractions, and automatically generated their detection algorithms using predefined code templates. However, this code smell detection focuses on the analysis of source code structures, and disregards the platform design information. Thus, it often cannot identify PSCSs.

Cong et al. [3] proposed a rule-based approach to code smells discovery for performance analysis. Each rule is composed of performance metrics that are collected from existing performance tools and compilers. Since some rules are based on the metrics from runtime information such as runtime memory access analysis (intercepting loads / stores to the buffer at runtime), it may take a long time to obtain the information. Accordingly, fast and efficient detection of PSCSs based on static code analysis is desired.

3 PSCS Alert System

This section first describes the overview of the proposed PSCS alert system, and then discusses the PSCS description and the process of PSCS detection in detail.

Our PSCS alert system takes an AST-based approach to representing code smells in a formal style with a clear hierarchy. PSCS patterns are defined in a formal and standard format using XML based on AST information. Figure 1 shows the overview of our PSCS alert system. We assume that the system first parses the original application code to build an AST [24]. This instance of an AST is then converted to an XML document whose elements correspond to Fortran language constructs. In our system, XPath is used for pattern matching to identify potential PSCS patterns in the XML document. Finally, the list of identified PSCS patterns is displayed for users. The PSCS alert system provides an interactive user interface to allow a user to use her/his knowledge for limiting the number of identified patterns, which need to be manually checked. The static analyses performed over ASTs and XML trees allow the proposed system to detect PSCSs efficiently before the application execution.

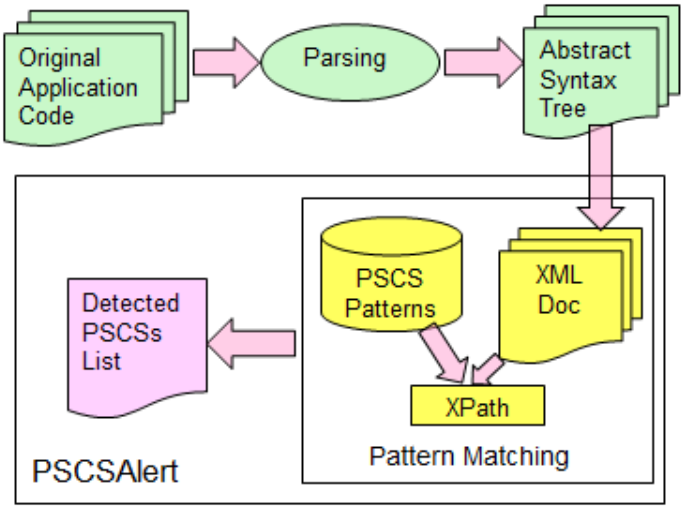


Figure 1: Framework of the proposed PSCS alert system.

```
!$acc kernels loop
do i = 1, M
  do j = i, N     !Here  is the triangular loop
    A(i,j) = i + j
  enddo
enddo
```

Figure 2: Loop body for Source Code of a Triangular Loop.

3.1 Platform-Specific Code Smell Description

In this work, we focus on typical PSCSs that are found in real applications [25, 26] and given by expert knowledge [27, 28]. Considering OpenACC programming [29], PSCSs used in this paper are listed in Table 1. Those code patterns are obviously harmful for platforms, in which OpenACC is used in GPU programming. Therefore, some code refactorings such as simply avoiding the code pattern, i,e., eliminating the code smell, are required to improve the performance portability across multiple platforms including the OpenACC-based platform.

It is not easy to directly find the PSCS patterns in the source code. Using an AST, hence, the identification of a specific code pattern is considered as a tree matching. In the system, an AST is represented as an XML document because XML is a standard way to express a data tree. By utilizing XML, various XML-related technologies are available to handle the AST.

Some of the PSCSs in Table 1 are considered as variants of the triangular loop pattern. Hence, we use the triangular loop as an example to explain how to derive our PSCS specifications. Figure 2 shows the source code of a typical triangular loop that is considered as a PSCS. In this source code, index variable i is used to determine the lower bound of the inner loop. When using OpenACC to parallelize this loop nest for GPUs, the OpenACC compiler copies out the entire A array from device to host and in the process copies garbage values into the lower triangle of the host copy of A [27]. OpenACC users should be aware of triangular loops, even though they may not degrade the performance in other platforms. Accordingly, a triangular loop is a PSCS.

Figure 3 shows the tree-structured XML document that represents the AST of the triangular loop in Figure 2. The XML document in Figure 3 can be illustrated as a tree shown in Figure 4. In Figure 4, the subtree enclosed in the dotted frame represents a triangular loop PSCS pattern. Therefore, if this pattern is found in an AST, we can find a PSCS.

```
<FortranDo style="0" end="1" nlabel="" slabel="">
 <AssignOp>
  <VarRefExp name="i"/>
  <IntVal value="1"/>
 </AssignOp>
 <VarRefExp name="M"/>
 <NullExpression/>
 <BasicBlock>
  <FortranDo style="0" end="1" nlabel="" slabel="">
   <AssignOp>
    <VarRefExp name="j"/>
    <VarRefExp name="i"/>
   </AssignOp>
   <VarRefExp name="N"/>
   <NullExpression/>
   <BasicBlock>
    <ExprStatement>
     <AssignOp>
      <PntrArrRefExp>
       <VarRefExp name="A"/>
        <ExprListExp>
         <VarRefExp name="i"/>
         <VarRefExp name="j"/>
        </ExprListExp>
      </PntrArrRefExp>
      <AddOp>
       <VarRefExp name="i"/>
       <VarRefExp name="j"/>
      </AddOp>
     </AssignOp>
    </ExprStatement>
   </BasicBlock>
  </FortranDo>
 </BasicBlock>
</FortranDo>
```

Figure 3: XML of Triangular Loop in Figure 2.

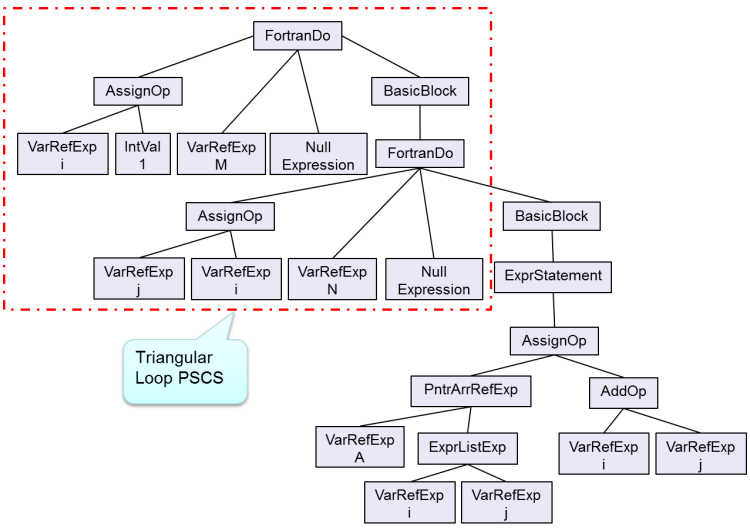


Figure 4: The XML tree structure for the triangular loop in Figure 2. The subtree enclosed in dotted frame indicates a triangular loop PSCS.

Table 1: Platform-Specific Code Smells

PSCS Name	Description
Triangular Loop	The number of iterations of the inner loop depends on the value of the outer loop's loop variable.
Live-out Scalars	A scalar variable updated within a loop is reused or referenced after the loop is called a "live-out" scalar, because correct execution may depend on the last value it was assigned in a serial execution of the loop(s).
Once-used Array Data	Some loops will fail to offload because parallelization is inhibited by arrays that must be privatized for correct parallel execution. In an iterative loop, data that is used only during a particular iteration but is not initialized prior to the region and is re-initialized prior to any use after the region can be seen as a once-used array data.
Computed Index	Computed index may be used for computations on multi-dimensional arrays that have been linearized. If the original loop with a computed index into the linearized array is compiled, compiler will give information like "Parallelization would require privatization of array."
Variable Length Loop	The length of the loop is variable, not known until run.
Common Subexpression	There is another occurrence of the expression whose evaluation always precedes this one. Such an expression is a common subexpression.
Loop Invariant	If a quantity is computed inside a loop during every iteration, and its value is the same for each iteration, it can vastly improve efficiency to hoist it outside the loop and compute its value just once before the loop begins.

3.2 PSCS Pattern Matching Process

XPath is the W3C standard language for expressing traversal and navigation in XML trees. This paper uses XPath expression to specify PSCS patterns in a source code. In this subsection, we will explain the pattern matching process using XPath expressions for a typical PSCS, Variable Length Loop.

The nested DO loops shown in Figure 5 are considered as Variable Length Loop PSCS because the length of the inner loop is not known in advance; i.e., the bound expressions are computed by loop statements within its outer loop or depend on the index variable of its outer loop.

To detect the above situations with one detection algorithm, we assume that the loop length is variable and the loop thus contains Variable Length Loop PSCS if any index variable or any updated variable is used in bound expressions of the inner loops.

We select all the variable reference expressions in a particular loop nest using the following XPath expression.

```
//FunctionDefinition/BasicBlock/FortranDo[1]//VarRefExp
```

We select all the loop control variable reference expressions of the specified loop nest using the following expression. Here, symbol "." selects the current node.

```
//FunctionDefinition/BasicBlock/FortranDo[1]/(./AssignOp/VarRefExp[1]
|.//FortranDo/AssignOp/VarRefExp[1])
```

We select all the detected variable expressions in the specified loop body using the following XPath expression.

```
//FunctionDefinition/BasicBlock/FortranDo[1]//ExprStatement//VarRefExp
```

Hence, we select all the bound expressions of loops by removing loop control index variable expressions and variable expressions in the loop body from all variable reference expressions using the following XPath expression.

```
!situation 1
DO 200 M=1,MF
 DO 200 K=1,KF
  DO 200 J=1,JF
   DO 200 I=1,INUM
    DO 200 L=lstart,lend
      ......
200 CONTINUE

!situation 2
DO M=1,3
 DO K=1,KF
  DO J=1,JF
    DO I=1,INUM
     DO L=lstart,lend
       ......
   ENDDO
  ENDDO
 ENDDO
ENDDO
```

Figure 5: The Variable Length Loop PSCS code sample in Numerical Turbine [25].

```
//FunctionDefinition/BasicBlock/FortranDo[1]/(.//VarRefExp
except ((./AssignOp/VarRefExp[1]|.//FortranDo/AssignOp/VarRefExp[1])
|.//ExprStatement//VarRefExp))
```

Predicate "except" selects all of a given node set, except for certain nodes.

Then, we select all the left-hand side variable expressions in loop statements and loop control index variable expressions using the following XPath expression.

```
//FunctionDefinition/BasicBlock/FortranDo[1]//(.//AssignOp/VarRefExp[1]
|.//ExprStatement/AssignOp/PntrArrRefExp/VarRefExp)
```

To check if the index variables or left-hand side variables in loop statements appear in the bound expressions of loop nest, we use the following XPath expression.

```
(//FunctionDefinition/BasicBlock/FortranDo[1]/(.//VarRefExp except
((./AssignOp/VarRefExp[1]|.//FortranDo/AssignOp/VarRefExp[1])
|.//ExprStatement//VarRefExp)))[@name =//FunctionDefinition/BasicBlock
/FortranDo[1]//(.//AssignOp/VarRefExp[1]|.//ExprStatement/AssignOp
/PntrArrRefExp/VarRefExp)/@name]
```

It returns the loop bound expressions if the bound expression has the same "name" attribute with a left-hand side variable. If the above XPath expression returns a non-empty node set, there is a Variable Length Loop PSCS in the specified loop nest.

We use the "for" statement to check each loop nest in the XML document of the source code using the following XPath expression.

```
for $i in //FortranDo,
$j in $i/(.//VarRefExp except ((./AssignOp/VarRefExp[1]
   |.//FortranDo/AssignOp/VarRefExp[1])|.//ExprStatement//VarRefExp)),
$k in $i //(.//AssignOp/VarRefExp[1]
   |.//ExprStatement/AssignOp/PntrArrRefExp/VarRefExp)
return (if($j/@name=$k/@name) then $j else null)
```

We can define the Variable Length Loop PSCS using the above XPath expression. Definitions of the seven PSCSs are shown in Table 2.

XML is a well-designed and widely used data format for the representation of a data structure such as an abstract syntax tree. The XML-based pattern matching allows us to describe PSCS patterns in a formal way using XPath expressions, and thus detect PSCS patterns in a systematic way. The AST information represented in an XML format enables the proposed pattern matching to use syntactic information so as to produce more accurate results. XPath provides fast XML searching processing. With our PSCS alert system, existing PSCSs of a program can be detected correctly and quickly before actually running the program. Use of such a standard technology is very important to assure the future-proofness of the code pattern representation. Hence, the proposed system can be robust during the long life span of an HPC application. The abbreviated syntax of XPath is more compact than its full syntax, and allows XPath to be written and read easily using intuitive and, in many cases, familiar characters and constructs. The full syntax of XPath is more verbose, but allows for more options to be specified, and is more descriptive. Thus, compared with other techniques in the literature, users can define their own patterns using XPath expressions with less effort.

Table 2: Definitions of PSCSs using XPath expressions

PSCS Name	XPath Expression
Triangular Loop	for $i in //FortranDo, $j in $i//FortranDo/(VarRefExp[1]|AssignOp/VarRefExp[2]) return $j[@name=$i/AssignOp/VarRefExp[1]/@name]
Live-out Scalars	for $i in //FortranDo, $j in //ExprStatement/AssignOp/VarRefExp, $k in $i//PreprocessingInfo/text() return $j[not(contains($k, $j/@name))]
Once-used Array Data	for $i in //FortranDo, $j in $i//ExprStatement, $k in $j, $m in $j/AssignOp/PntrArrRefExp/VarRefExp, $n in $k/AssignOp//VarRefExp, $l in $i//PreprocessingInfo/text() return (if($m/@name=$n/@name[not(contains($l, 'private'))]) then $m else null)
Computed Index	for $i in //FortranDo, $j in $i//ExprStatement/AssignOp/VarRefExp[1]/@name, $k in $i//ExprStatement/AssignOp/PntrArrRefExp//VarRefExp/@name return (if($j=$k) then $k else null)
Variable Length Loop	for $i in //FortranDo, $j in $i/(.//VarRefExp except ((./AssignOp/VarRefExp[1]|.//FortranDo/AssignOp/VarRefExp[1])| .//ExprStatement//VarRefExp)), $k in $i //(.//AssignOp/Var RefExp[1] \|.//ExprStatement/AssignOp/PntrArrRefExp/VarRefExp) return (if($j/@name=$k/@name) then $j else null)
Common Subexpression	for $i in //FortranDo//FortranDo[not(.//FortranDo)], $j in $i//ExprStatement/AssignOp//(MultiplyOp|AddOp|SubtractOp| DivideOp), $k in ($i//ExprStatement/AssignOp//(MultiplyOp|AddOp|SubtractOp|DivideOp) except $j) return (if(deep-equal($j, $k) and count($j//(MultiplyOp|AddOp|SubtractOp|DivideOp))>=2) then ($j, $k) else null)
Loop Invariant	for $i in //FortranDo//FortranDo[not(.//FortranDo)], $j in $i//ExprStatement, $k in $j/following-sibling::*, $m in $j/AssignOp/(MultiplyOp|AddOp|SubtractOp|DivideOp), $n in $k/AssignOp/(MultiplyOp\|AddOp\|SubtractOp\|DivideOp) return(if (deep-equal($m, $n) and count($m/AssignOp/(MultiplyOp|AddOp |SubtractOp|DivideOp)>=2)) then $m else null)

4 Evaluation

In this section, we first evaluate the effectiveness of the proposed alert system, and then evaluate performance changes before and after eliminating these PSCSs. In practical uses, various code

refactorings will be applied to the code portions of detected PSCSs for high performance portability. A typical example is to use #ifdef to change the code portions to be compiled for individual platforms. However, in the performance evaluation, a code is refactored so as to simply eliminate detected PSCSs and to make the code portions not harmful to an OpenACC-based platform. This will allow the platform to achieve a higher performance. The purpose of the performance evaluation is to demonstrate that the detected PSCSs really affect the performance of an OpenACC-based platform and hence the proposed system can alert performance-sensitive code patterns in large-scale application codes.

4.1 Evaluation of the PSCS Alert System

In Section 4.1, we evaluate the effectiveness of the proposed alert system using seven PSCSs and two real applications, Numerical Turbine [25] and MSSG-A (Multi-Scale Simulator for the Geoenvironment - Atmosphere Model) [26]. Those two applications have been developed for a vector supercomputer, NEC SX-9. In their source codes, thus, there exist a lot of PSCSs harmful to OpenACC-based platforms.

We assume that when the AST of a source code has an identical subtree with a predefined AST of the PSCS, the source code is considered to have a PSCS. We validate the proposed alert system from the following aspects:

1. Comprehensive (recall). The predefined pattern should work for general use. All true PSCSs in application codes that have been predefined by our PSCS alert system should be detected, that is to say, the PSCS alert system has a recall of 100%.

2. Correctness (precision). Considering the trade-off between precision and recall, we assume that the detection method is better than the random choice one. Therefore, we assume that 50% of the detected PSCSs should be the true PSCSs with respect to recall 100%.

3. Impact. By eliminating the detected PSCSs, the modified application should perform better than the original one to utilize the target platforms.

We validate the results of the proposed alert system by analysing the pre-identified PSCSs and detected PSCSs in the context of the systems. The pre-identified PSCSs have been identified by the authors and experts who have migrated real applications to GPU computing platforms based on the application code reading and profiling results. Validation of the detected PSCSs is performed by application engineers to identify whether a detected PSCS is actually a pre-identified PSCS or not. As with the work of Moha [23], the measures of precision and recall are introduced to evaluate the proposed alert system. We use the recall to assess the number of detected PSCSs among the pre-identified PSCSs, and use the precision to assess the number of PSCSs identified among the detected PSCSs. Equations (1) and (2) are used to calculate the recall and precision.

$$recall = \frac{|\{Detected\ PSCSs\}|}{|\{Pre-identified\ PSCSs\}|}. \tag{1}$$

$$precision = \frac{|\{Identified\ PSCSs\}|}{|\{Detected\ PSCSs\}|}. \tag{2}$$

4.1.1 Precision and Recall on Numerical Turbine

We perform the evaluation of the proposed alert system on Numerical Turbine [25], which is a 3-D flow calculation application. Variable Length Loop PSCS listed in Table 1 appears frequently in a subroutine named EXPLICIT of Numerical Turbine, which consumes 53.49% of the total execution time. We applied the XPath expression for detecting Variable Length Loop PSCS illustrated in Section III to the XML document transformed from the source file.

Table 3 presents the recall and precision of the detection of the PSCSs listed in Table 1 in the kernel loops of Numerical Turbine. In the table, "Pre-identified PSCSs" refers to the PSCSs in the

Table 3: Precision and Recall on Numerical Turbine

PSCSs	Pre-identified PSCSs	Detected PSCSs	Identified PSCSs	Precision (%)	Recall (%)
Triangular Loop	0	0	0	-	-
Live-out Scalars	0	0	0	-	-
Once-used Array Data	0	0	0	-	-
Computed Index	1	1	1	100	100
Variable Length Loop	13	13	13	100	100
Common Subexpression	0	0	0	-	-
Loop Invariant	0	0	0	-	-

Table 4: Precision and Recall on MSSG-A

PSCSs	reisnre2				
	Pre-identified PSCSs	Detected PSCSs	Identified PSCSs	Precision (%)	Recall (%)
Variable Length Loop	5	4	4	100	80
Common Subexpression	0	0	0	-	-
Loop Invariant	0	0	0	-	-
Computed Index	6	9	6	66.7	100
PSCSs	tracer				
	Pre-identified PSCSs	Detected PSCSs	Identified PSCSs	Precision (%)	Recall (%)
Variable Length Loop	25	25	25	100	100
Common Subexpression	23	7	7	100	30.4
Loop Invariant	7	7	7	100	100
Computed Index	0	0	0	-	-

application code that have been identified by authors and experts, "Detected PSCSs" refers to the PSCSs detected by the proposed alert system, and "Identified PSCSs" refers to the detected PSCS that is identified to be a pre-identified PSCS. The PSCS pattern matching process is finished in less than three seconds, even though the source file consists of 16K lines of code.

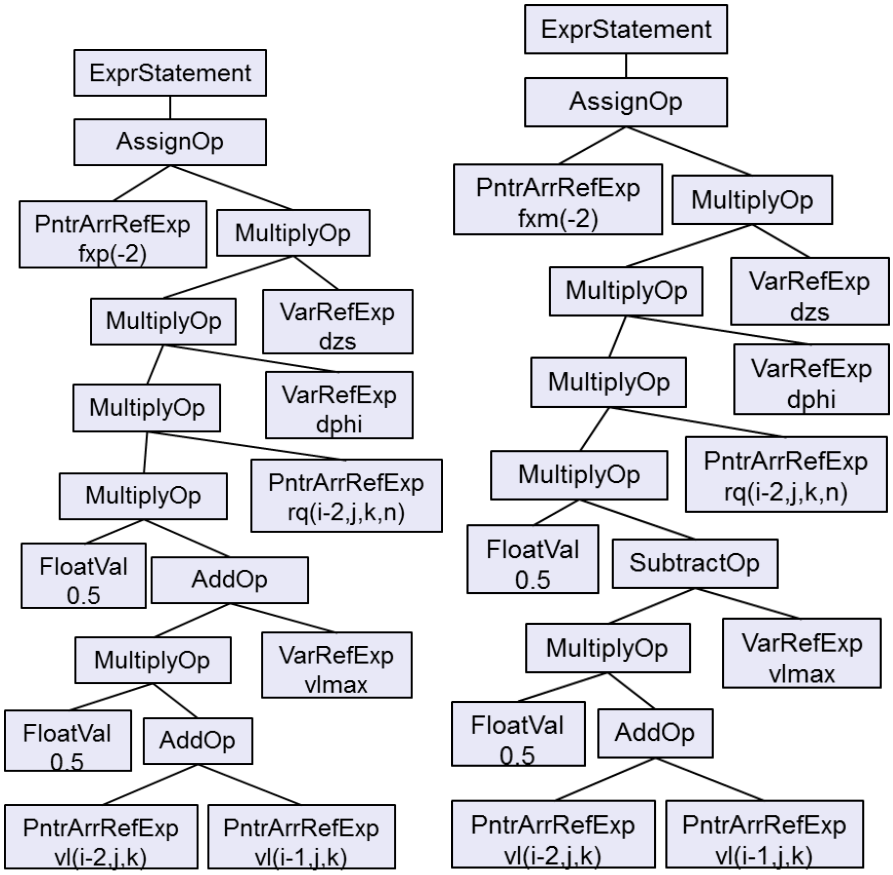
The recall ratio of our alert system is 100% for each PSCS. We specify the XPath expression for each PSCS for general use and assess its impact on precision. The Variable Length Loop PSCS exists in 13 loops while the entire kernel contains 15 loops. We get a perfect precision ratio of 100% for Variable Length Loop PSCS. The evaluation results in Table 3 clearly show that the proposed PSCS alert system also has high precision ratios for the other PSCSs existing in the application. This indicates that our predefined PSCS patterns are general enough for practical uses.

We use "-" to indicate for those PSCSs that are not found in Numerical Turbine. Other six predefined PSCSs are not detected in this application.

4.1.2 Results on MSSG-A

Table 4 shows the precision and recall of the PSCSs listed in Table 1 on MSSG-A [26]. MSSG is a coupled non-hydrostatic atmosphere-ocean-land global circulation model. It has been developed for the purpose of promoting advanced projection/prediction simulation. The MSSG application contains overall 276K lines of code, of which 189K lines form the atmosphere model part, called MSSG-A. We applied the proposed PSCS alert system on two subroutines (*reisnre2* and *tracer*) of MSSG-A that take most of the computation time.

As shown in Table 4, four kinds of PSCSs are detected in the MSSG-A subrotuines and the proposed alert system can achieve high precision and recall ratios for most of PSCSs in MSSG-A. Therefore, these results clearly demonstrate that XPath can correctly express most PSCSs.



(a) Example AST for the first assignment (b) Example AST for the second assignment)

Figure 6: Example ASTs with subtree forms a clone

A single exception is the recall for Common Subexpression, which is as low as only 30.4%. This is because the AST-based detection can only find the identical subtree, but cannot find clone codes when the subtrees or nodes that are derived from the same root forms clone codes. As an example, suppose the following assignment statements and their corresponding ASTs in Figure 6.

$$fxp(-2) = 0.5\_DP * (0.5\_DP * (vl(i-2,j,k) + vl(i-1,j,k))$$

$$+vlmax) * rq(i-2,j,k,n) * dphi * dzs.$$

$$fxm(-2) = 0.5\_DP * (0.5\_DP * (vl(i-2,j,k) + vl(i-1,j,k))$$

$$-vlmax) * rq(i-2,j,k,n) * dphi * dzs.$$

Then, we can see that there are two common subexpressions existed in the two assignment statements:

$$0.5\_DP * (vl(i-2,j,k) + vl(i-1,j,k)),$$

$$0.5\_DP * rq(i-2,j,k,n) * dphi * dzs.$$

However, the AST-based detection takes the codes that have identical subtrees as common subexpressions, and hence finds only

$$0.5\_DP * (vl(i-2,j,k) + vl(i-1,j,k)),$$

as Common Subexpression PSCS, while obviously

$$0.5\_DP * rq(i-2,j,k,n) * dphi * dzs,$$

is also a Common Subexpression PSCS.

Even if all the pre-identified PSCSs can be identified based on code reading and profiling results, the alert system may fail in identifying some of the pre-identified PSCSs because of the limitation of the AST-based method used in the proposed system. The AST-based detection method misses some of code clones, which are formed by several AST nodes derived from the same root. The proposed system does not deform an expression. Therefore, it cannot find common subexpressions that appear as a result of expression deformation. To find such a subexpression, more advanced analysis will be required as discussed in [30].

4.1.3 Changes in Compiler Messages by Eliminating PSCSs

The results shown in Sections 4.1.1 and 4.1.2 demonstrate that the proposed PSCS alert system has high precision and recall ratios for detecting predefined PSCSs. As a PSCS potentially degrades the performance portability, removal of the PSCS is expected to improve the performance on other platforms, for which the original application code is not optimized. We evaluate the impact of eliminating the detected PSCSs on the performance of real applications by observing the compiler messages of the two real applications. The compiler used in the evaluation is the PGI compiler 12.10, and the "-acc" option is used to enable OpenACC directives on GPU accelerators. CUDA 5.0 and NVIDIA GPU Tesla C2070 with compute capabilities 2.0 are used in the experiment. Table 5 shows the observation results.

In Table 5, each row presents the code modification and compiler information changes for a specific PSCS. Column "Original code snippets" shows that the code contains a PSCS, and the column "Modified code snippet" shows the code where a PSCS is eliminated. The elimination of detected PSCSs in a large-scale application code would require in-depth knowledge of the platform and the application. Since we already know how to eliminate these predefined PSCSs, we build a database to store the transformation recipe of eliminating PSCSs to help users remove PSCSs. The detected PSCSs can be eliminated by applying the transformation recipes to the XML documents of the application codes. The transformation recipes for the elimination of five PSCSs: Triangular Loop, Live-out Scalars, Once-used Array Data, Computed Index and Variable Length Loop are provided in this work. The transformation recipes for the elimination of Common Subexpression and Loop Invariant PSCSs are not provided since common subexpression elimination (CSE) and loop-invariant code motion are usually performed by compilers. However, as a compiler performs an optimization when the cost / benefit analysis proves that it is worthwhile. Common Subexpression and Loop Invariant PSCSs still exist in the application code in some cases. Therefore, we will explore the general transformation recipes for eliminating Common Subexpression and Loop Invariant PSCSs in the future work.

Table 5: Compiler message changes before and after eliminating PSCSs

PSCS	Original code snippet	Compiler message	Modified code snippet	Compiler message
Triangular Loop	!$acc kernels loop do i=1,M do j=i,N A(i,j)=i+j end do end do	(Execution results may be wrong) Loop is parallelizable Accelerator kernel generated Loop is parallelizable	!$acc kernels loop copy(A) do i=1,M do j=i,N A(i,j)=i+j end do end do	Loop is parallelizable Accelerator kernel generated Loop is parallelizable

table continued on next page

continued from previous page

PSCS	Original code snippet	Compiler message	Modified code snippet	Compiler message

Live-out Scalars	!$acc kernels loop
	do i = 1, N
	do j = 1, N
	do k = 1, N
	idx = B(i,k)*C(k,j)
	end do
	A(i,j) = idx
	enddo
	enddo
	B(1,1)=idx
		Loop is parallelizable
		Inner sequential loop scheduled on accelerator
		Accelerator kernel generated
		Accelerator restriction: induction variable live-out from loop: idx
			!$acc kernels loop
			do i = 1, N
			!$acc do private(idx)
			do j = 1, N
			do k = 1, N
			idx = B(i,k)*C(k,j)
			end do
			A(i,j) = idx
			enddo
			enddo
			B(1,1)=idx
				Loop is parallelizable
				Loop is parallelizable
				Accelerator kernel generated

Once-used Array Data	!$acc kernels loop
	do i = 1, M
	do j = 1, N
	do jj = 1, 10
	tmp(jj) = jj
	end do
	A(i,j) = sum(tmp)
	enddo
	enddo
		Parallelization would require privatization of array 'tmp(:)'
		Sequential loop scheduled on host
		Loop is parallelizable
		Accelerator kernel generated
			!$acc kernels loop
			do i = 1, M
			!$acc loop private(tmp)
			do j = 1, N
			do jj = 1, 10
			tmp(jj) = jj
			end do
			A(i,j) = sum(tmp)
			enddo
			enddo
				Loop is parallelizable
				Loop is parallelizable
				Loop is parallelizable
				Accelerator kernel generated
				Loop is parallelizable
				Accelerator kernel generated

Computed Index	!$acc kernels loop do i=1,M do j=1,N idx=((i-1)*M)+j A(idx)=B(i,j) end do end do	Parallelization would require privatization of array 'a(:,:1)' Accelerator scalar kernel generated Parallelization would require privatization of array 'a(:,:1)'	!$acc kernels loop do i=1,M do j=1,N A(i,j)=B(i,j) end do end do	Loop is parallelizable Loop is parallelizable Accelerator kernel generated
Variable Length Loop	DO 200 M=1,MF DO 200 K=1,KF DO 200 J=1,JF DO 200 L=lstart,lend II1 = IS(L) II2 = II1+1 IIF = IT(L) DO 200 I=II2,IIF 200 CONTINUE	Loop carried reuse of 'qll' prevents parallelization Loop carried reuse of 'qrr' prevents parallelization Inner sequential loop scheduled on accelerator	DO 200 M=1,MF DO 200 K=1,KF DO 200 J=1,JF DO 200 I=1,INUM DO 200 L=lstart,lend 200 CONTINUE	Loop is parallelizable Accelerator kernel generated

table continued on next page

continued from previous page

PSCS	Original code snippet	Compiler message	Modified code snippet	Compiler message
Common Subexpression	!$acc kernels DO m=1,MF DO k=1,JF DO j=1,KF DO i=1,INUM uxi(i,j,k,m)= (qrr(i,j,k,m)+ u(i,j,k,m))*u(i,j,k,m) *dphi*dzs*j uxi(i,j,k,m)= (qll(i,j,k,m)- u(i,j,k,m))*u(i,j,k,m) *dphi*dzs*j ENDDO ENDDO ENDDO ENDDO !$acc end kernels	(# of computation in innermost loop for each iteration in opencl kernel) Int: 34 add, 24 multiply Double: 2 add, 8 multiply	!$acc kernels DO m=1,MF DO k=1,JF DO j=1,KF DO i=1,INUM tmp=u(i,j,k,m)*dphi *dzs*j uxi(i,j,k,m)= (qrr(i,j,k,m)+ u(i,j,k,m))*tmp uxi(i,j,k,m)= (qll(i,j,k,m)- u(i,j,k,m))*tmp ENDDO ENDDO ENDDO ENDDO !$acc end kernels	(# of computation in innermost loop for each iteration in opencl kernel) Int: 29 add, 21 multiply Double: 2 add, 5 multiply
Loop Invariant	!$acc kernels DO m=1,MF DO k=1,JF DO j=1,KF DO i=1,INUM uxi(i,j,k,m)= (qrr(i,j,k,m)+ u(i,j,k,m))*u(i,j,k,m) *dphi*dzs*j uxi(i,j,k,m)= (qll(i,j,k,m)- u(i,j,k,m))*u(i,j,k,m) *dphi*dzs*j ENDDO ENDDO ENDDO ENDDO !$acc end kernels	(# of computation in innermost loop for each iteration in opencl kernel) Int: 32 add, 24 multiply Double: 4 add, 8 multiply	!$acc kernels DO m=1,MF DO k=1,JF DO j=1,KF tmp=dphi*dzs*j DO i=1,INUM uxi(i,j,k,m)= (qrr(i,j,k,m) +u(i,j,k,m))*u(i,j,k,m) *tmp uxi(i,j,k,m)= (qll(i,j,k,m)- u(i,j,k,m))*u(i,j,k,m) *tmp ENDDO ENDDO ENDDO ENDDO !$acc end kernels	(# of computation in innermost loop for each iteration in opencl kernel) Int: 32 add, 24 multiply Double: 2 add, 4 multip

The original program with Triangular Loop PSCS may generate wrong computation results. This is because when the compiler copies out the entire array data from the device to the host, it copies garbage values into the lower triangle of the host copy in this process. An OpenACC copy clause can be specified on the accelerator region boundary to guarantee that the correct code is generated.

By removing Live-out Scalars, the compiler message is changed from "Accelerator restriction: induction variable live-out from loop: idx" to "Loop is parallelizable." When the scalar variable "idx" is used for debugging purpose, privatizing this scalar variable will be helpful to generate a much more efficient, fully-parallel kernel. However, we have to notice that the value printed out for "idx" in the print statement will be different from that in a sequential execution of the program.

The compiler message for the program that contains Once-used Array Data changed from "Parallelization would require privatization of array 'A(:)', sequential loop scheduled on host" to "Loop is parallelizable." Arrays that are initialized prior to any use after the region can be declared private. An OpenACC directive "!$acc do private()" can provide the compiler with the information necessary

to successfully compile the nested loop into a fully parallel kernel for execution on a GPU.

The compiler messages of the code snippet in Numerical Turbine that contains Variable Length Loop PSCS are also illustrated in Table 5. This application is originally developed for NEC SX-9. When running it on GPUs, the variable length loop becomes a PSCS and it has to be eliminated. By eliminating PSCSs, the compiler message changes from "Accelerator scalar kernel generated" to "Loop is parallelizable." This compiler message change indicates that we can better utilize the accelerator by executing the application in parallel and thus improve the performance.

Common Subexpression PSCS occurs multiple times in MSSG-A. Common subexpression elimination refers to the process of detecting multiple occurrences of the same computation and replacing them with a single occurrence. Instead of generating a code to compute the common subexpression two times, the compiler generates the code, places the result in a register, and reuses it. When compiled with the PGI compiler, the compiler messages for the code snippets before and after common subexpression elimination are the same. To check whether the generated kernel changed or not, we use CAPS Many-core Compiler (Version 3.3.4) with the option "hmpp –openacc -target OpenCL gfortran -O3" to look into the detail of the generated OpenCL kernel. We compared the numbers of arithmetic operators in the innermost loop for each iteration before and after eliminating PSCSs. The statistics data show that the arithmetic operators after eliminating the common subexpression are reduced, which means a benefit when the application scales.

We also observed the generated OpenCL kernel of the code snippets in MSSG-A that contain the Loop Invariant PSCS. We eliminate the Loop Invariant PSCS by hoisting the loop invariant out of the innermost loop, and then obtain a significant reduction in the number of arithmetic operators in the innermost loop for each iteration.

These compiler messages and OpenCL kernel information demonstrate that the PSCS patterns predefined in the proposed alert system actually change the codes generated by the OpenACC compiler, and hence are performance-sensitive. Accordingly, our alert system is useful to find performance-sensitive code patterns in large-scale application codes.

4.2 Effects of Eliminating PSCSs on Performance Portability of Applications

To prove that the proposed alert system can actually identify the PSCSs that degrade performance portability, we have examined the impact of eliminating the detected PSCSs on the performance of real applications, Numerical Turbine and MSSG-A, by measuring the speedup ratios of code excerpts before and after eliminating PSCSs. Note that the three PSCSs, Triangular Loop, Live-out Scalars and Once-used Array Data can be detected using the proposed alert system. However, they do not exist in these two real applications. Therefore, test programs are used in the evaluation.

4.2.1 Experimental Setup

Real applications, Numerical Turbine and MSSG-A, are originally optimized for a vector system. In the evaluation, we examine the performance changes on other systems by eliminating detected PSCSs. Test programs in [27] are used to evaluate the PSCSs that do not appear in the two real applications. The system configurations used in the following evaluation are listed in Table 6.

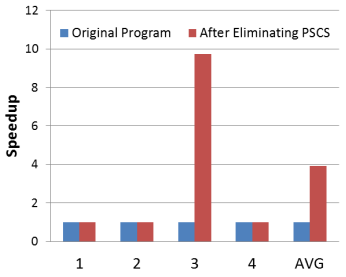
4.2.2 Experimental Results and Discussion

The evaluation results are shown in Figure 7. The vertical axis shows the speedup ratio against the original program after eliminating PSCSs. The horizontal axis shows the platforms listed in Table 6. The speedup ratio of the *Original Program* on each platform is 1.

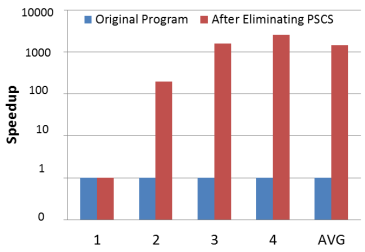
Figure 7 shows that the performance of Platform 1 is almost unchanged by eliminating Triangular Loop, Live-out Scalars, Once-used Array Data, Common Subexpression and Loop Invariant. On the other hand, the performance of each GPU platform is usually increased by their elimination, even though eliminating Variable Length Loop reduces the performance of Platform 2 in the evaluation. From these results, it is obvious that detected PSCSs are code patterns that clearly affect the performance, and hence eliminating those PSCSs leads to performance improvement of GPUs.

Table 6: System Configurations

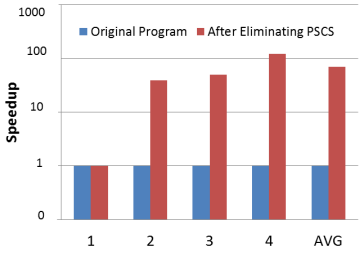
Platform	CPU	GPU	Compiler
1	Intel Xeon CPU E5-695 v2	–	Intel compiler ifort 14.0.2
2	Intel Core i7 920	NVIDIA Tesla C1060	PGI compiler 12.10
3	Intel Core i7 930	NVIDIA Tesla C2070	PGI compiler 12.10
4	Intel Core i7 920	NVIDIA Tesla K20	PGI compiler 12.10



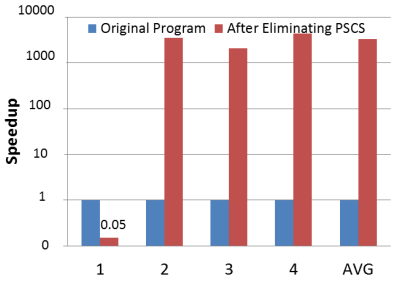
(a) Results for Triangular Loop.



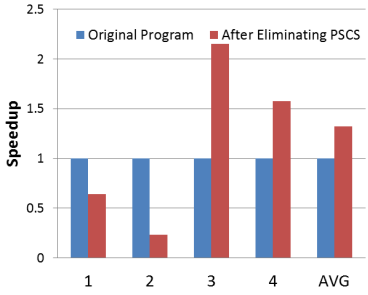
(b) Results for Live-out Scalars.



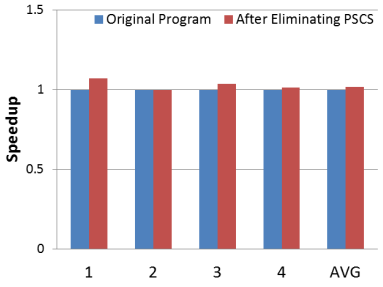
(c) Results for Once-used Array Data.



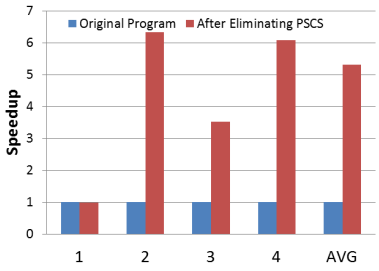
(d) Results for Computed Index.



(e) Results for Variable Length Loop.



(f) Results for Common Subexpression.



(g) Results for Loop Invariant.

Figure 7: Speedup ratios by eliminating PSCSs.

Although the seven PSCSs do not usually affect the performance of Intel CPUs, they have to be eliminated for the platforms using OpenACC for NVIDIA GPUs. Therefore, the proposed system can alert users to performance-sensitive code patterns in large-scale applications. The code portions with detected PSCSs can be refactored to improve the performance portability across multiple platforms.

As shown in Figures 7(d) and (e), eliminating Computed Index PSCS and Variable Length Loop PSCS lead to performance degradation on the original platform that they are optimized for, even though performance improvement could be obtained on other platforms with different GPUs. With those code patterns, it is difficult to keep the performance portability among multiple platforms. OpenACC is designed for code portability across multiple platforms, but the performance is not actually portable. Thus, some additional mechanisms are required to make the performance portable. For example, an extensible programming framework named Xevolver [24] can evolutionarily improve the application so as to have high performance portability without messing up the original code, because it can transform an application code in different ways for individual platforms. The Xevolver could be a promising tool to develop applications with high performance portability.

5 Conclusions

In this paper, we present a PSCS alert system for detecting the code patterns of platform-specific code smells by using the syntactic information represented by XML-based ASTs. PSCSs that have clear patterns could be expressed and thus identified by the proposed alert system. The evaluation results using real applications demonstrate that the proposed PSCS alert system has high recall and precision ratios for detecting PSCSs referred to in this research. Accordingly, it is clear that an XML-based approach is one portable and promising way to describe PSCS patterns.

As we already know how to remove a PSCS, we build a database of the transformation recipes for eliminating the predefined PSCS patterns with the help of Xevolver. The transformation recipes are written in XSLT files. Thereby, the proposed alert system can give users suggestions of how to remove the detected PSCSs by applying the transformation recipes to the XML files of the application codes. The general transformation recipes for the elimination of Common Subexpression and Loop Invariant PSCSs will be discussed in our future work. The performance evaluation results demonstrate that the detected PSCSs really affect the performance of an OpenACC-based platform and hence the proposed system can alert performance-sensitive code patterns in large-scale application codes. Therefore, the proposed system provides a good starting point for improving performance portability.

Acknowledgment

This work was partially supported by JST CREST "An Evolutionary Approach to Construction of a Software Development Environment for Massively-Parallel Heterogeneous Systems" and Grant-in-Aid for Scientific Research (B) # 25280041. The authors are grateful to Prof. Ryusuke Egawa and Prof. Kazuhiko Komatsu for providing instance list for optimizing HPC applications. The authors also thank Prof. Satoru Yamamoto and Dr. Keiko Takahashi for providing their application codes, Numerical Turbine and MSSG.

References

[1] Martin Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Longman Publishing Co. Inc., 1999.

[2] Tom Mens and Tom Tourwé. A survey of software refactoring. *Software Engineering, IEEE Transactions on*, 30(2):126–139, 2004.

[3] Guojing Cong, I-Hsin Chung, Hui-Fang Wen, D. Klepacki, H. Murata, Y. Negishi, and T. Moriyama. A systematic approach toward automated performance analysis and tuning. *Parallel and Distributed Systems, IEEE Transactions on*, 23(3):426–435, March 2012.

[4] Tim Bray, Jean Paol, C. M. Sperberg-McQueen, Eve Maler, and Franois Yergeau. Extensible markup language (XML) 1.0 (fifth edition). `http://www.w3.org/TR/REC-xml/`, November 2008.

[5] A. Berglund, S. Boag, D. Chamberlin, M. Fernández, M. Kay, J. Robie, and J. Siméon. Xml path language (xpath) 2.0 (second edition). `http://www.w3.org/TR/xpath20/`, January 2011.

[6] Michael Kay. Xsl transformations (xslt) version 3.0, w3c last call working draft 2. `http://www.w3.org/TR/xslt-30/`, October 2014.

[7] M. Mantyla, J. Vanhanen, and C. Lassenius. A taxonomy and an initial empirical study of bad smells in code. *International Conference on Software Maintenance, 2003(ICSM 2003)*, 381-384, 2003.

[8] Garcia Joshua, Popescu Daniel, Edwards George, and Medvidovic Nenad. Identifying architectural bad smells. In *Proceedings of the 2009 European Conference on Software Maintenance and Reengineering*, CSMR '09, pages 255–258, Washington, DC, USA, 2009.

[9] Jacek Ratzinger and Michael Fischer. Improving evolvability through refactoring. In *In MSR 05: Proceedings of the 2005 international workshop on Mining software repositories*, pages 1–5, 2005.

[10] H.P. Putro and I Liem. Xml representations of program code. In *Electrical Engineering and Informatics (ICEEI), 2011 International Conference on*, pages 1–6, July 2011.

[11] H.P. Putro. Code smell detection in source code in ast representation with by rules detection approach. Master's thesis, Institut Teknologi Bandung, Bandung, Indonesia, 2011.

[12] Stefan Slinger. Code smell detection in eclipse. Master's thesis, Delft University of Technology, 2005.

[13] R.C. Holt. Structural manipulations of software architecture using tarski relational algebra. In *Reverse Engineering, 1998. Proceedings. Fifth Working Conference on*, pages 210–219, Oct 1998.

[14] G. Travassos, F. Shull, M. Fredericks, and V.R. Basili. Detecting defects in object-oriented designs: Using reading techniques to increase software quality. *Proc. 14th Conf. Object Oriented Programming, Systems, Languages, and Applications*, pages 47–56, 1999.

[15] A. Apostolico and Z. Galil. Pattern matching algorithms. Technical report, Oxford University Press, UK, 1997.

[16] W. Yang. Identifying syntactic differences between two programs. *Software-Practice and Experience*, pages 739–755, 1991.

[17] J. J. Hunt and W. F. Extensible language-aware merging. In *Software Maintenance, 2002. Proceedings. International Conference on*, pages 511–520, 2002.

[18] Iulian Neamtiu and Including Bind. Understanding source code evolution using abstract syntax tree matching. In *Proceedings of the 2005 international workshop on Mining software repositories*, pages 2–6, 2005.

[19] F.A. Fontana, M. Zanoni, A. Marino, and M.V. Mantyla. Code smell detection: Towards a machine learning-based approach. In *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, pages 396–399, Sept 2013.

[20] Pmd. `http://pmd.sourceforge.net/`, 2014.

[21] Nikolaos Tsantalis and Alexander Chatzigeorgiou. Identification of extract method refactoring opportunities for the decomposition of methods. *Journal of Systems and Software*, 84(10):1757 – 1782, 2011.

[22] C. Marinescu, R. Marinescu, P. F. Mihancea, and R. Wettel. iplasma: An integrated platform for quality assessment of object-oriented design. In *In ICSM (Industrial and Tool Volume*, pages 77–80. Society Press, 2005.

[23] N. Moha, Y. Gueheneuc, L. Duchien, and A Le Meur. Decor: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering*, 36(1):20–36, 2010.

[24] Hiroyuki Takizawa, Shoichi Hirasawa, and Hiroaki Kobayashi. Xevolver: an xml-based programming framework for software evolution. *SC13*, November 2013.

[25] Satoshi Miyake, Satoru Yamamoto, Yasuhiro Sasao, Kazuhiro Momma, Toshihiro Miyawaki, and Hiroharu Ooyama. Unsteady flow effect on nonequilibrium condensation in 3-d low pressure steam turbine stages. In *In ASME Turbo Expo 2013*, volume 5B: Oil and Gas Applications; Steam Turbines, San Antonio, Texas, USA, June 2013.

[26] Takahashi Keiko, Azami Akira, Tochihara Yuki, Kubo Yoshiyuki, Itakura Ken'ichiKen'ichi, Goto Koji, Kataumi Kenryo, Takahara Hiroshi, Isobe Yoko, Okura Satoru, Fuchigami Hiromitsu, Yamamoto Jun-ichi, Takei Toshifumi, Tsuda Yoshinori, and Watanabe Kunihiko. World-highest resolution global atmospheric model and its performance on the earth simulator. In *State of the Practice Reports*, SC '11, pages 21:1–21:12, New York, NY, USA, 2011. ACM.

[27] The Portland Group PGI. 11 tips for maximizing performance with openacc directives in fortran. `https://www.pgroup.com/resources/openacc_tips_fortran.htm`, 2013.

[28] Ryusuke Egawa. An hpc refactoring catalog; guidelines to bridge the gap between hpc systems. In *Special Session: Legacy HPC Application Migration 2013 (LHAM) (held in conjunction with IEEE MCSoC-13)*, Tokyo, September 27 2013.

[29] Michael Wolfe. The openacc application programming interface, version 2.0. `http://www.openacc-standard.org/`, June, 2013.

[30] Rainer Koschke, Raimar Falke, and Pierre Frenzel. Clone detection using abstract syntax suffix trees. In *Reverse Engineering, 2006. WCRE'06. 13th Working Conference on*, pages 253–262, 2006.