Degree project

Adaptation of Legacy Codes to Context-Aware Composition using Aspect-Oriented Programming for Data Representation Conversion

Author: Alisa Sotsenko
Supervisor: Antonina Danylenko

Date: 2013-02-17

Course Code: 5DV00E, 30 credits
Level: Master

Department of Computer Science
Abstract

During the software developing process new functional and non-functional requirements emerge. Therefore, such software systems should adapt their behaviour according to changes in the program or execution environment.

Different computation problem domains such as sorting, matrix multiplication, etc. usually require different data representations and algorithms variants implementations in order to be adapted and re-designed to context-aware composition (CAC). Context-aware composition is a technique for the design of applications that can adapt its behavior according to changes in the program. We considered two application domains: matrix multiplication and graph algorithms (DFS algorithm in particular). The main problem in the implementation of the representation mechanisms applied in these problem domains is time spent on the data representation conversion that in the end should not influence the application performance.

One of the solutions to such a problem is adaptation to CAC approach with aspect-orientated programming (AOP) that allows adapting software behaviour according to the current context at runtime (e.g. the number of processors available or the problem size). We use AOP approach with data representation conversion to increase the application performance.

This thesis work presents a flexible aspect-based architecture that includes the data structure representation adaptation in order to reduce implementation efforts required for adaptation different application domains.

Although, manual approach has small overhead 4-10% for different problems compared to the AOP-based approach, experiments show that the manual adaptation to CAC requires on average three times more programming effort in terms of lines of code than AOP-based approach. Moreover, the AOP-based approach showed the average speed-up over baseline algorithms that use standard data structures of 2.1.

Keywords: aspect-oriented programing, context-aware composition, matrix multiplication, graph algorithm, data representation
List of Figures

Figure 2.1  Object-oriented design pattern for adaptation to CAC. .......................................................... 4
Figure 3.1  Design of profiling/learning phase in CAC .......................................................... 10
Figure 3.2  Design of the Composition Block with AOP approach to CAC ........................................... 11
Figure 3.3  Offline learning with AOP .......................................................................................... 12
Figure 4.1  Design of the collection all algorithm implementation variants ........................................ 13
Figure 4.2  Package structure of our implementation ................................................................... 13
Figure 4.3  Design of the context parameters and data representation parameters .................. 14
Figure 4.4  Design of the generator for input values of algorithm implementation variant
.................................................................................................................................................. 15
Figure 4.5  Profiling block implementation .................................................................................. 16
Figure 4.6  Sequence diagram of profiling block ......................................................................... 17
Figure 4.7  Learning block implementation ................................................................................... 17
Figure 4.8  Design the conversion of data representation variants .............................................. 18
Figure 4.9  Sequence diagram of composition block ................................................................... 18
Figure 4.10 Composition block implementation with AOP ......................................................... 19
Figure 4.11 Model for Matrix Multiplication ............................................................................... 19
Figure 4.12 Model for Graph algorithms .................................................................................... 22
Figure 5.1  Execution time of Baseline and CAC approaches for Matrix Multiplication (M1)
.................................................................................................................................................. 26
Figure 5.2  Execution time of Baseline and CAC approaches for Matrix Multiplication (M2)
.................................................................................................................................................. 26
Figure 5.3  Execution time of Alev and CAC approaches for Graph algorithms (M1) .......................... 27
Figure 5.4  Execution time of Alev and CAC approaches for Graph algorithms (M2) .................... 28
List of Tables

Table 5.1 Speed-up of different CAC approaches for MM with data representations... 27
Table 5.2 Speed-up of different CAC approaches for Graph Algorithm (in msec). ...... 28
# Table of content

1 Introduction.................................................................................................................. 1  
   1.1 Introduction to the problem....................................................................................... 1  
   1.2 Motivation and Research goals................................................................................ 1  
   1.3 Thesis outline........................................................................................................ 2  
2 Context-Aware Composition......................................................................................... 3  
   2.1 Prerequisites for Adaptation to Context-Aware Composition................................. 3  
   2.2 Online learning ..................................................................................................... 4  
   2.3 Offline learning...................................................................................................... 5  
   2.4 Offline learning with CAC ..................................................................................... 6  
   2.5 Summary............................................................................................................... 7  
3 Context aware composition with Aspect oriented programing................................. 8  
   3.1 Aspect-oriented programming............................................................................... 8  
   3.2 Concerns of Context-Aware Composition with AOP ........................................... 9  
      3.2.1 Offline learning with Aspect-Oriented Programming ..................................... 11  
4 Implementation .......................................................................................................... 13  
   4.1 Implementation details .......................................................................................... 13  
   4.2 Matrix multiplication algorithms .......................................................................... 19  
      4.2.1 Profiling and Learning Blocks with AOP ......................................................... 20  
      4.2.2 Composition block with AOP ........................................................................ 21  
      4.2.3 Summary ....................................................................................................... 21  
   4.3 Graph algorithms .................................................................................................. 22  
      4.3.1 Profiling and learning with AOP .................................................................... 22  
      4.3.2 Composition block with AOP ....................................................................... 23  
      4.3.3 Summary ....................................................................................................... 24  
   4.4 Conclusions .......................................................................................................... 24  
5 Experiments ................................................................................................................ 25  
   5.1 Experimental set-up............................................................................................... 25  
   5.2 Matrix multiplication algorithms .......................................................................... 25  
   5.3 Graph algorithms ................................................................................................ 27  
   5.4 Summary .............................................................................................................. 28  
   5.5 Lines of Code ...................................................................................................... 29  
6 Related work ............................................................................................................... 30  
7 Conclusions and future work ................................................................................... 31  
   7.1 Project evaluation summary .................................................................................. 31  
   7.2 Future work ......................................................................................................... 31  
8 References................................................................................................................... 33
1 Introduction

In this chapter we gives short overview about the problem domain, describes research goal and motivation of this master thesis.

1.1 Introduction to the problem

One of the important requirements for software systems is maintenance [14], where the software requires modification over time as new functional or non-functional requirements [6] or changes in run-time environment happen. In situations when changes in requirements or in run-time environment properties occur frequently in the short time term [6], it can be beneficial for the software systems to have flexible architecture that allows reconfiguration, context-awareness [5], autonomic adaptation [14,15], etc. One of the approaches to develop such systems is context-aware composition approach [5] that allows adapting system’s behavior in accordance to changes in the context information or environment. Context information is the information about the system environment (e.g., the number of available processors, memory, network connection), the sensors information (e.g., current location, time). This information can be represented by generators inside of program, as for example, the problem size and data structure representation for matrix multiplication.

During development process a software developer or system designer often encounters a problem of choosing the best-fit component variant for the current context at runtime. Best-fit component variant is variant that optimizes software by improving performance, etc. Component variant is the one of the possible algorithms that perform computing tasks which are time consuming and computational resources (e.g., sorting, matrix operations). Furthermore the best-fit component variant depends not only on algorithm implementation but on data representation variant. Data representation variant is used in component variant and can be represented by different structure types (e.g., arrays, collections, lists) with different elements data types (e.g., int, real). Data representation can affect on the performance of component variant as different structures have different elements access time, etc. and we do not know which data representation will be best-fit for each component variant with current context information. Moreover data representations conversion to the best one takes additional time that should not affect on the application performance in general.

In order to reduce redesign efforts for making software adaptable to changes in run-time environment or system context, we chose the aspect-oriented approach as base for our approach that dynamically selects best-fit data representation variant with respect to the chosen best-fit component variant with regard to the current context information. We suggest an extension based on the approach that was represented in work [1] to provide flexible architecture for conversion data representations and consider another test case, Graph algorithms.

1.2 Motivation and Research goals

The research goal of this work is to reduce implementation efforts, in terms of lines of code, required for adaptation of the existing legacy code to context-aware composition. Moreover, we aim to increase the application performance by selecting the best-fit component variant in terms of a data type representation and an algorithm. The main idea is to use aspect-oriented programming approach for context-aware composition adaptation.
Different computation problem domains such as sorting, matrix multiplication, etc. usually require different data representations and algorithms variants implementations in order to be adapted and redesigned to context-aware composition.

In this thesis we consider two application domains such as matrix multiplication and graph algorithm problems. The first problem is that significant amount of time is spent on the matrix multiplication of large size. Therefore it is unknown beforehand which multiplication algorithm will execute faster for different context information (matrix size, etc.). The second problem is that a lot of time is spent on depth-first search (DFS) in large graph. Therefore, it is unknown in advance which data representation for graph will be best for different graph algorithm implementation variants.

The main problem in implementation of mechanism representation is time spending on data representation conversion that should not affect on the application performance. In this thesis we consider aspect-oriented programming as an appropriate technique to adapt existing legacy code for different tasks without redesign and reimplementaton of software. We want to show that this approach should make software more reusable and improve application’s performance.

1.3 Thesis outline

The remainder of this thesis is as follows: In Chapter 2 we present an overview of context-aware composition mechanism and different approaches for online and offline learning. Chapter 3 presents the implementation issues of our approach with aspect-oriented programming. Chapter 4 gives a detailed description of the implementation extension to approach with two problem scenarios: matrix multiplication and graph algorithms. Chapter 5 evaluates this approach, describes experiments setup and results. In Chapter 6 we discuss related work. Finally, Chapter 7 concludes this thesis and presents the future work.
2 Context-Aware Composition

Context-aware computation is an essential part of a wide range of application domains, where an application should adapt its behavior according to potentially changing context or environment properties during its execution. Context-oriented programming is a technique for the design of such applications. Context-aware composition (CAC) as a special case of context-aware computation aims at adapting applications to changing call contexts and available resources in the system environment [1].

CAC is a technique to design the ubiquitous computing systems. Context information contains information about system’s dynamic environment, as for example, available memory for a task or number of available processors. During the parallel task execution the context information can change. Operating memory and number of available processors can be different for different tasks. Context-awareness allows reacting to the changing context information that provides suitable solution for the current context.

Context-aware compositions operate on the following concepts:

- **The legacy code** considered in this work is matrix multiplication and graph algorithms.
- **Context:** the problem size, number of available processors that can be used in the system.
- **Component variant:** algorithm and data representation that is used in this algorithm.

The context-aware composition mechanism requires information about which variant to bind to a certain variation point for a certain context to best satisfy an optimization goal (e.g., performance or memory efficiency) [6]. This information can be represented in multidimensional array or stored in dispatch table [2,4] and collected after learning phase [4].

CAC can improve software systems by dynamically composing the best variant of alternative algorithms, data structures, availability computational resources for each dynamic composition context [1]. CAC adapts already existing applications with dynamic changing context. Hence implementation of new applications along with CAC can be time consuming and requires additional efforts for changing application design. Therefore applications have to comply with some prerequisites that are described in next section.

2.1 Prerequisites for Adaptation to Context-Aware Composition

According to the work presented in [1] the object-oriented design pattern is required for adaptation of legacy code to CAC. As shown in Fig. 2.1 all component variants must implement the same interface and design, where each component is implemented as abstract data type ADT that encapsulates states and algorithms operating with data representations of that state. The abstract Representation class has abstract method that changes the data to appropriate representation before a corresponding call to an algorithm variant occurs. The algorithm variants are implemented separately from a representation variant (the abstract Algorithm class) that makes components reusable and flexible by changing them independently. For instance, for sorting arrays the algorithm variants could be Selection Sort, QuickSort, Merge Sort and data representations are Array, List, ArrayList, Collection with Integer, Double, Real etc. type of elements.
Before each call to the algorithm variant, the best-fit data representation variant should be selected and the call to the convertor method \textit{clone()} should occur in order to change representation. Moreover, the current context information with current algorithm variant should be taken into account in order to choose the most appropriate data representation variant. One of the problems during the design of context-awareness systems is detection of best fit variant for the current context. One of the approaches is learning. Learning is a technology that is focused on the design and development of the algorithms that allow systems to automatically extract information or learn structure from the given data [4]. There are two learning strategies: online and offline. Offline learning is a model that allows systems to learn structure from the static dataset and requires memory for storing and reconsidering previous examples. Online learning is a model that allows systems to learn structure with sequentially incoming data in one pass without reconsidering previous examples.

2.2 Online learning

Learning is a technology that is focused on the design and development of the algorithms that allow systems to automatically extract information or learn structure from the given data [4].

Learning operates on these concepts:

\textit{Training data} is data incoming from different sources like artificial data, sensory data, databases, human generated data etc.

\textit{Training instance} is a portion of training data that represents some object or class.

\textit{Decision function} is a function which makes decision with respect to the input data.

\textit{Expert advice} is set of decision rules, decision function or any predictable learning algorithm.

\textit{Supervised learning} is machine learning with labelled instances [19]

\textit{Batch learning} is supervised machine learning with decision function for training data which maps instances to label set, one of the example is offline learning.

Online Learning algorithm observes a stream of training instances at system runtime. This technique corresponds to supervised learning and proceeds in a series of trials. At each trial \( t \) the prediction algorithm receives a next observation \( a \) and computes the
prediction for this observation \( c' = \text{sign}(W_c \cdot a) \), thereafter a correct outcome \( c \) is revealed. Thus the algorithm seeks a prediction that minimizes the number of mistakes that occur if \( c \neq c' \) over a worst case sequence of \( a \) and \( c \) [23]. More formally, at each trial the online learning algorithm takes as input a hypothesis \( h \) that enrolls weight vector \( W_c \) and a training instance \((a, c)\) and returns a new hypothesis \( \hat{h} \) [27]. The objective is to update the weights \( W_c \leftarrow W_{c-1} \) using a certain update rule that has to include the information provided of the current instance \((a, c)\) combined with current hypothesis \( h \) to produce a new hypothesis \( \hat{h} \) [21]. A typical example of such online learning with feedback is a spam filter in e-mail box, when the filter identifies a spam e-mail based on the examples that are received every time a user specifies a spam-e-mail.

There exist two different tendencies in online learning that have been studied in the theoretical machine learning literature: (1) predicting from expert advice and (2) online learning from example [22]. The former problem requires a certain set of experts \( E \) that predict a possible decision \( c' \) for an actual context \( a \). Using this approach at each trial after the algorithm receives an instance \( a \), (2) it receives a prediction from experts \( E \), (2) then the algorithm makes its own prediction based on the outcome of \( E \), and, thereafter, (3) the correct decision \( c \) is revealed. The natural objective of such approach is to predict nearly as well as the best expert. The intuition behind is to construct a so-called master algorithm that feeds the given observations to the experts \( E \), and then uses some function \( f \) of the \( N \) experts predictions to construct its own prediction [23]. Such algorithms differ in a penalization function, i.e. how to penalize the experts that give wrong prediction (e.g. to discard all inconsistent expert in each trial [23], or to penalize each mistaken expert by multiplying its weight [24]), calculation of an expected loss (e.g. Euclidean distance), and the function applied to derive the prediction based on expert advice (e.g. using weighted majority [24]).

The latter problem of online learning from example can be considered as a more general scenario, in which the algorithm also proceeds in trials, however it removes the expert advises and focuses on the prediction function \( f \). Similarly to the previous approach, throughout the trials an algorithm learns a so-called hidden vector \( W_c \) that should properly give the correct decision and minimize the prediction loss [25]. Each time when a prediction mistake is made the algorithm is penalized. Therefore, the objective of this approach is to make as few mistakes as possible.

### 2.3 Offline learning

Offline learning is one of the most widely used learning approaches in conventional machine learning applications, such as classification and regression. The learner is provided with a labelled decision relation and its task is to construct a decision function that maps an input instance \( a \) (usually not included into prior decision relation) to the output decision \( c \). This means the learner is first trained by exposing it to a labelled decision relation, so that it can learn association between the input (contexts) and output (decisions), usually referred to as hypothesis, and establish a decision function. Typical decision models used in supervised learning are: Artiﬁcial Neural Networks, Decision Trees, Bayesian Networks, Support Vector Machines, etc. [26]. Among the popular applications of supervised learning are: pattern recognition including speech recognition, image recognition, and spam detection. A key issue with supervised learning is the choice of training set, that can lead to sufﬁciently effective learning. Supervised learning is suitable for the problems where training data can be easily prepared and arranged. Such training data is either retrieved from past history of the system or is predictable in future.
In context of autonomic computing, supervised learning can be used to learn or determine most suitable actions mapped to the different states of a system. The learning methods mentioned above will scale well to the autonomic systems where selection of actions (adaptation) can be generalized as a problem of classification or regression.

2.4 Offline learning with CAC

Offline Learning implies batch learning, that is, a decision function is constructed from the batch of training instances that are currently available. To apply offline learning the self-adaptive system requires to address the following concerns: offline profiling, decision function to extrapolate or interpolate the profiling results, runtime analysis, component (or action) selection and dynamic composition (if needed) for actual context and system environment. The separation of the former concerns from actual system functionality can be performed manually [6] or automatically (e.g. using Aspect-Oriented Programming) [1].

The offline learning phase implies that the component variants (or actions) are tested for different actual contexts and the champion variant of each context is determined. This information is extrapolated and interpolated using decision function (e.g. decision trees, support vector machines, etc.) from a total mapping of contexts to champion variants. This decision function is used in the actual composition phase for selecting the presumably optimal variant for each actual context. Offline learning can be used as a reflective sub-system in self-adaptive system if the following prerequisites are established:

- the formal context, i.e. the set of attributes that influence the system's behaviour, is established and will not change over time;
- the sample of actual contexts is a correct representative of all possible variants;
- the alternative component variants (possible actions) are known and will not change over time.

All these prerequisites identify that there is no uncertainty in our knowledge about the system's behaviour. However such prerequisites are usually not the case in self-adaptive systems in most of the scenarios. Long training time, explicit training data, and inability to perform under uncertainty are some of the key issues with offline learning approach. It can require learning time ranging from few seconds to many hours, and even weeks in case of complex systems. If changes in system behaviour occur at runtime (e.g. changes in the formal context or environment that make the constructed decision function not valid representative of the best-fit system behaviour), long training time will be a problem in case of real time systems. Decision relation is another problem in case of offline training. If it is not a good representative of the real time data, then runtime system performance will be low. On the other hand, a basic learning requirement for self-adaptive systems is that the learning process must be fast enough, dynamic and continuous. The system should not close or stop its learning process, rather it should adapt or learn its behaviour in response to the changing context and other system properties. This means that in case of self-adaptive autonomic systems, we have to switch our learning approach from offline to online training methods, or at least combine them, i.e. allow the system change its decision function due to unexpected changes in the environment.

Decision information is usually represented by a decision model. A decision model is a set of rules that determines a target decision and can be implemented as decision tables, decision trees, decision graphs, dispatch tables[2,4]. The selection of an appropriate decision model has an impact on application performance in terms of memory consumption and execution time. Choosing an appropriate decision model for a specific task requires a trade off between memory consumption, accuracy, and speed
for construction and utilization. Additionally, every decision model requires a set of operations for processing decision information to adjust and improve application behaviour with respect to specific domain requirements.

2.5 Summary
CAC allows adopting existing application depending on the changes in the environment properties or in the system itself at run-time without reengineering or refactoring existing code. Hence the applications have to comply with prerequisites that are considered in this chapter. For training phase the test data needs to be generated by sample generator. Online learning can handle the changes in the environment. However, along with the benefits of quick in-time learning, a major drawback of online learning is that it will add to the workload on a live system. Therefore there is the need to look for lightweight methods of online machine learning to be applied to self-adaptive systems. Therefore in this work we concentrate on offline learning solution with dispatch table decision model. To design the CAC with data representation and to reuse it with new graph algorithms application we use aspect-oriented programming which is described in next section.
3 Context aware composition with Aspect oriented programing

In this chapter we introduce the aspect-oriented programing with examples and show how we can use this technique to adapt legacy code. After that we describes in detail the main component of Template pattern[1] to adapt legacy code and benefits of using aspect oriented approach for offline learning.

3.1 Aspect-oriented programming

Aspect-oriented programming (AOP) is an effective module approach used in software engineering to separate crosscutting concerns from primary concerns [5]. The crosscutting concerns that are implemented as aspects are parts of program that that affect many other parts of the system. Primary concerns are the main application code as for example matrix multiplication application, etc. AOP allows to abstract the crosscutting code into separate aspects and then apply the code dynamically where the pointcuts are defined in the main code. The pointcuts defines where an aspect should be applied during the application compilation or execution. Moreover, AOP allows adding new functionality to application without any knowledge about it or additional changes in the main code [3].

The AOP includes concepts which are defined below:

Aspect: a modularization of a concern that cuts across multiple classes. Aspect in Java is regular class with @Aspect annotation and may have methods, fields as any other class:

```java
@org.aspectj.lang.annotation.Aspect
public class myAspect {
    //class body
}
```

Join point: is code that runs during the execution of a program, such as the execution of a method or the handling of an exception. As for example, anyMethod() and doAccessCheck() are join point.

```java
@org.aspectj.lang.annotation.Aspect
public class myAspect {
    @Pointcut("execution(public * *(..))")
    private void anyMethod() {} 

    @Before("com.example.dataAccessOperation()")
    public void doAccessCheck() {
        // ...
    }
}
```

Advice: action taken by an aspect at a particular join point. This is additional code that can be executed "around," "before" and "after" join point.

Pointcut: a predicate that matches join points. Advice is associated with a pointcut expression and runs at any join point matched by the pointcut (for example, the execution of a method with a certain name). Pointcut in Java is method with @Pointcut annotation with expression when it run, as for example bellow the method anyMethod() will be execute when the any public method will be invoked:

```java
@Pointcut("execution(public * *(..))")
private void anyMethod() {}
```
**Before advice**: Advice that executes before a join point, but which does not have the ability to prevent execution flow proceeding to the join. Advice in Java is a method with @Before annotation and expression specifying when the advice will be run. In example below the method `doAccessCheck()` will be run before the `dataAccessOperation()` method run:

```java
@Before("com.example.dataAccessOperation()")
public void doAccessCheck() {
    // ...
}
```

**After advice**: Advice that executes after a join point. After advice is declared using the @Aspect annotation and runs after the method return that is specified in advice expression:

```java
@After("com.example.dataAccessOperation()")
public void doReleaseResources() {
    // ...
}
```

**Around advice**: Advice that surrounds a join point such as a method invocation. Around advice runs “around” join point. As for bellow example, instead of `dataAddOperation()` method execution the method `execute()` will be executed:

```java
@Around("com.example.dataAddOperation ()")
public Object execute(ProceedingJoinPoint pjp) throws Throwable {
    //body...
}
```

Around advice can perform custom behaviour before and after the method invocation. It is also responsible for choosing whether to proceed to the join point or to shortcut the advised method execution by returning its own return value or throwing an exception. For example, this kind of advice is used for recursive methods invocation.

The feature of AOP is dividing an application into the main part and the aspect part that allows adopting legacy code by adding the aspect part without reengineering the main part of application [5]. For example, AOP has weaving mechanism that allows combining programs at runtime, replacing method body implementation to another one or executing additional code before, after and around method body. AOP approach improves applications performance and reusability by optimization of main code.

### 3.2 Concerns of Context-Aware Composition with AOP

Considering the prerequisites which are defined in a previous chapter 2 the next concern can be identified: the data conversion mechanism should be implemented according to object-oriented design pattern, and define the best-fit data representation variant with respect to the method implementation variant. The data conversion mechanism provides the method to convert the data structure representation to another one. Additionally the large number of data representation variants with different conversion methods can be presented in application implementation.

For the learning phase, the representation data generator needs to be implemented with abstract change representation method that should be implemented manually for each application. Moreover the context parameters (as the problem size, available processors and data structure representation) should be also generated by parameter generator. Once the context parameters are generated, the performance of methods execution and data conversation is measured for each implementation variant and the best-fit representation variant with implementation variant is defined with respect to the
current context parameters. The dispatch table stores the best representation and implementation variant for each context parameters.

For the **composition phase**, the dispatch method with aspect advice needs to be implemented. First, the best-fit data structure representation is chosen and the conversion is performed if needed. Second, the best algorithm implementation variant is invoked.

To make the CAC implementation reusable for other applications (as Sorting, Matrix multiplication, etc.), it is necessary to implement profiling and learning phases separately from composition phase to avoid changing legacy code. We use the already designed Template pattern (Fig.3.1) [1]. This pattern implements the common behaviour of profiling/learning for all applications that have established prerequisites for adaptation to CAC.

![Diagram of profiling/learning phase in CAC](image)

The Template consists of three blocks:

- **Profiling block** is responsible for binding the current context with optimal implementation variants. In this block all implementation variants are invoked in different context parameters (problem size, available processors, data representation, etc.). Class `Profiler` generates context parameters and training data for implementation variants. Two abstract methods `generateContext()` and `generateTrainData()` should be implemented for each ADT. During of profiling process the time of conversion with execution implementation variant is measured for each context. The time functions can be implemented as measurement functions, using mathematical models for the approximations, or a combination of functions [8]. The best data representation variant with implementation variant for each context are passed to the learning block. Profiling is a process that collects the knowledge for context-aware composition and defines inefficient or ineffective data representation variant for implementation variant.

- **Learning block** learns a decision function using the set of best data representation variants and algorithm implementation variants with context parameters. In our work, the dispatch table is used for mapping from each context to the best data representation
and algorithm implementation variants. The Decision Function class has two abstract methods that Dispatch Table class implements. The decide() method selects the best-fit data representation with algorithm implementation variant for current context parameters in dispatch table and then passes to the decider() method in Composition block after that the best-fit algorithm implementation variant will execute.

– **Composition block** is responsible for optimization of application execution by combining combination the best data representation with algorithm implementation variant during the learning and training phase at runtime. The AOP approach is used with pointcut on method call as shown in the Fig. 3.2 and around advice that has access to the current context parameters and dispatch table to get the best-fit data representation and algorithm implementation variant.

![Figure 3.2 Design of the Composition Block with AOP approach to CAC](image)

During the profiling phase the weaving mechanism is used and the pointcut methodCall is specified in composition block. Aspect with around advice runs instead of algorithm implementation variant. The decider() method returns best data representation with algorithm implementation variant.

### 3.2.1 Offline learning with Aspect-Oriented Programming

The offline learning with CAC was described in section (2.4). In this section we describe the offline learning with AOP.

One of the benefits of using AOP approach during the offline learning is the knowledge from previous learning iteration that could be used in current learning phase. Hence, it is beneficial if the recursion is presented in the implementation variant.
As shown on Fig. 3.3 the aspect around advice interrupts each call implementation variant for selecting the best data structure representation from dispatch table. The called conversion method returns the data in a new data structure representation. When we have the data in the best representation variant the aspect selects the best algorithm implementation variant from dispatch table for the current context parameters and invokes it instead of the current variant. During the aspect execution the time spend on the data conversion and implementation variant execution is measured, and the lowest time value identifies the best data representation with implementation variant. After the best variants are saved in dispatch table, the new context parameters are generated and passed to the all implementation variants.

AOP approach allows adapting the existing code without reengineering or redesigning it by adding the aspect with advices. For the CAC the aspect provides the application behavior or reaction on the context changes.
4 Implementation

In this chapter we describes the implementation details of our approach for both applications Matrix Multiplication and Graph algorithms.

4.1 Implementation details

The object-oriented Template pattern[1] that is described in chapter 3 implemented with Java/JDK 1.6 using AspectJ technology. Additionally we implemented the abstract class BaseMethodGenerator (Fig. 4.1) in Profiling block in order to provide flexible CAC architecture for adding new applications (Sorting algorithms, Matrix Multiplications, Graph algorithms, etc.). In our work we consider the implementations for Matrix Multiplication and Graph algorithms.

![Diagram of BaseMethodGenerator](image)

Figure 4.1 Design of the collection all algorithm implementation variants

The abstract method `getSourcePakcageName()` returns a path to the source code of each application, e.g. for MM the path is “applications.MatrixOperationsContext.algorithms” and for GA is “applications.GraphContext.algorithms”. The package structure Fig.4.2 of our implementation has package applications that contains sources of each application that need to be adapted.

![Package structure](image)

Figure 4.2 Package structure of our implementation

---

1 http://eclipse.org/aspectj/
The `getAllMethods()` method is responsible for automatically collecting algorithm implementation variants for each application by annotating each method of algorithm implementation. For instance, each algorithm implementation variant has annotations `@LogAction(order = 1, parallel = false)` and method `getAllMethods()` collect all this methods:

```java
Class[] classes = getClasses(getSourcePackageName());
// find all methods with annotations
for (Class cl : classes) {
    for (Method m : cl.getDeclaredMethods()) {
        if (m.isAnnotationPresent(LogAction.class)) {
            LogAction an = m.getAnnotation(LogAction.class);
            MethodInstance mi = new MethodInstance(null, null, null);
        }
    }
}
```

The `Parameter` class in the **Profiling block** is extended by additional abstract class `BaseParameterGenerator` (Fig. 4.3) with `generateAdditionalParameters()` abstract method that generates the possible data structure representation variants for learning phase. Therefore, this method returns the list of data representations `List<Parameter>` and can be extended by another parameters if a new application source code is required to be processed.

![Diagram](image)

**Figure 4.3 Design of the context parameters and data representation parameters**

The `generateContextParameters()` method generates all possible context parameters. For Matrix Multiplication this method generates different matrix sizes and numbers of available processors. For Graph algorithms it generates different numbers of graph node and available processors.

All possible data structure representation variants are generated by `generateAdditionalParameters()` before learning phase. After the context parameters are generated the input values for algorithm implementation variant are generated by `BaseInputParametersGenerator` (Fig. 4.4) class according to all possible context variants.
The parameters generated by `getInputParam()` method are used in **learning phase** in `train()` method of `Profile` class in **Profiling block** (Fig. 4.5). This is the abstract method and it should be implemented for each application as the input parameters in algorithm implementation variant for Matrix Multiplication and Graphs algorithms are different for each of them.

We considered the **learning** and **testing** phase during the **Profiling block** implementation. The learning phase is executed in two cycles: (1) all algorithm implementation variants with all generated context parameters are defined on first cycle while on the second one (2) each algorithm implementation variant is executed sequentially with all data structure representation variants. The best algorithm implementation and data structure representation variant are defined and saved to dispatch table after each iteration of the first cycle. Therefore, the `changeRepresentation()` method is executed in the second cycle and in the `train()` method of `BaseAspect` in **Composition block**.

The best algorithm implementation variant with data structure representation is defined after **learning phase** is processed. The learning results contain the execution time of each algorithm implementation variant with different data structure representation. Moreover, the number of best algorithm implementation variant and the number of best conversion are contained in learning results and saved in dispatch table. We present the pseudo code of learning phase in Algorithm 1.

Algorithm 1. The algorithm of learning phase

**Input:** generated test data, generated context parameters

**Output:** best algorithm implementation variant and data structure representation are saved in Dispatch table.

1: maxSize = 512, stride = 16, iterations = 512
2: for each parameter in GeneratedContextParameters do
3:   methods ← `getAlgorithmVariants(parameter)`
4:   for each size until maxSize with stride do
5:     for each method in methods do
6:       for each representation in RepresentationTypes do
7:         for each iteration in iterations do
8:           startTime ← `System.currentTimeMillis()`
9:           testData ← `changeRepresentation(representation)`
10:          method.invoke(testData)
11:        endTime ← `System.currentTimeMillis()`;

Figure 4.4 Design of the generator for input values of algorithm implementation variant
During the testing phase the context parameters are generated and passed to the TestRunner() class that is an extension of the Profile class. The additional parameters, such as data structure representation variants, are not generated on this phase and Representation class is not used in Profiling block. As best data structure representation variant is already known after learning phase than aspect in Composition block does conversion data representation to best one before the call of algorithm implementation variant at runtime. If the current data representation variant is best then the data conversion doesn’t occur. The time that is spent on the conversion is added to the algorithm execution time.

Figure 4.5 Profiling block implementation

The LearnerRunner() and TestRunner() classes in Profiling block are common for both Matrix Multiplication and Graph algorithm application. If the application supports algorithms with data representation then Representation class is used in learning and testing phases. If application does not support algorithms with data representation then Representation class doesn’t use in learning and testing phase by configure list of representations in generateAdditional Parameters() method of BaseParameterGenerator class. The sequence diagram of profiling block shown in Fig. 4.6.
The Learning block (Fig. 4.7) was extended by additional `BaseBestSolution` class that is responsible for loading learning results and for saving them in a dispatch table. There are two additional enums `ParameterType` and `RepresentationType` that are used to make code more understandable and reusable. The `ParameterType` enum consists of possible context parameters while the `RepresentationType` consist of all possible data representation types.

In the Composition block we implemented the `changeRepresentation()` abstract method of `Representation` class for Matrix Multiplication and Graph algorithms as shown in Fig. 4.8.
The sequence diagram of composition block shown in Fig. 4.9.

The Representation class is used in both the Profiler and the Composition block. In the Profiler block on the second cycle of learning phase the conversion of data representation is required as we learned all data structures to define the best one. In the Composition block (Fig.4.10) the Aspect is using changeRepresentation() method during the learning and testing phase to change current data representation to the best one. Additionally we provided the data representation conversion for both Matrix Multiplication and Graph algorithm application.
4.2 Matrix multiplication algorithms

We considered five algorithm implementation variants for matrix multiplication (MM) (Fig. 4.11)

4.2 Matrix multiplication algorithms

We considered five algorithm implementation variants for matrix multiplication (MM) (Fig. 4.11)

The variants are: **Baseline** – algorithm that reduces a matrix multiplication to a number of matrix - vector multiplications and then each such, in turn, to a number of vector multiplications; **Inline** – the standard implementation with three nested loops; **BaselineSparse** – a variation of baseline optimized for sparse matrices; **Recursive** – reduces a matrix multiplications to eight multiplications of eight sub-matrices; and **Strassen** – uses a recursive reduction with seven multiplications and uses threads for parallel multiplications [6]. All these variants have common interface that can be used for adaptation to CAC.

All these algorithms can be used with different data structure representation. In our case we have two types of data structures: dense matrices and sparse matrices. There are four possible data representation variants where first matrix is dense and second one
is dense (dense_dense), sparse_sparse, sparse_dense, dense_sparse. Density of a matrix is the ratio of none-zero elements ($O(N^2)$) in it and sparse matrix has a $O(N)$ none-zero elements. The context parameters for MM are the size of matrices, processors if are available and representation data structure.

4.2.1 Profiling and Learning Blocks with AOP

In this sub-section we describe in more details the implementation of components of Profiling and Learning blocks for MM with AOP. We implemented offline learning with AOP, which uses Aspect for changing data structure representation and choosing the best algorithm implementation variant for the current context parameters at runtime. Input parameters for algorithm implementation variant are generated by `getInputValues()` of `MatrixInputValueGenerator` then algorithm implementation variant of matrix multiplication invokes one by one with different data structure representations. After each method is invoked and the execution time is measured, the best data structure with algorithm implementation variant is defined. If the algorithm implementation variant has recursion or uses threads with recursion than recursion method call will be interrupted by aspect in order to select the best data representation with algorithm implementation variant, the representation will be changed if it is need and the selected best implementation will be invoked.

We present code snippet bellow of the actual code of the Recursive algorithm implementation variant for two matrices multiplication:

```java
(1) @LogAction(number = 3, isParallel = false)
(2) public double timesMatrix (Matrix<E> res, int resRowL, int resColL, Matrix<E> m1, int m1RowL, int m1ColL, int m1RowU, int m1ColU, Matrix<E> m2, int m2RowL, int m2ColL, int m2RowU, int m2ColU, boolean plus) {
    ...
    timesMatrix(..);
}
```

Each implementation of algorithms MM has annotation cf. line 1 `LogAction()` with two parameters: the number of algorithm implementation variant and true/false value that defines the parallel or the sequence algorithm execution. In `MatrixMethodGenerator` the package name “algorithms.MatrixMultiplication” is specified and the `getAllMethods()` is invoked in order to collect all algorithm implementation variants. From the snippet above 14 input method values of algorithm implementation variant for each context parameters are generated.

For generation context parameters we implemented two methods of `MatrixParameterGenerator` which are represented in the code snippet bellow:

```java
@Override
(1)public List<Parameter> generateParameters() {
(2)List<Parameter> parameterList = new ArrayList<Parameter>();
(3)parameterList.add(new Parameter(ParameterType.FIRST_DENSE, new Object[]{true}));
(4)parameterList.add(new Parameter(ParameterType.SECOND_DENSE, new Object[]{true}));
(5)parameterList.add(new Parameter(ParameterType.FIRST_CONTENT, new Object[]{true}));
(6)parameterList.add(new Parameter(ParameterType.SECOND_CONTENT, new Object[]{true}));
(7)parameterList.add(new Parameter(ParameterType.PARALLEL, new Object[]{false, true}));
```
In the `generateParameters()` method we generated five context parameters: `FIRST_DENSE` - the original data representation for first matrix; `SECOND_DENSE` - the original data representation for second matrix; `FIRST_CONTENT` - is responsible for the generation values in first matrix (more or less 0.0), `SECOND_CONTENT` - is responsible for generation values in second matrix (more or less 0.0), and last one is parallel or none-parallel algorithm implementation variant cf. line 7.

In the method `generatedAdditionalParameters()` two possible data representation variants `dense` or `sparse` cf. line 9-10 are used for each matrix.

### 4.2.2 Composition block with AOP

In this sub-section we describe the components implementation of the Composition block for MM with AOP and provide code examples.

During the learning phase for both Recursive and Strassen algorithm implementation variants the around advice of Aspect cf. line 1 is triggered before recursive method invocation. It selects the best solution, cf. line 4, that contains best-fit algorithm implementation variant for current context parameters `param` and data representation. Then it changes data representation variant, cf. line 6, to the best one and invokes best-fit variant for sub-matrix multiplication:

```java
@Around("@annotation(LogAction) && (@withincode(LogAction)) || @withincode(ThreadAnnotation))")
public Object training(ProceedingJoinPoint pjp) throws
IllegalArgumentException, InvocationTargetException,
IllegalAccessException {
    Map<Integer, BaseMethodsGenerator.MethodInstance> methods =
    methodsGenerator.getAllMethods(param);
    BestSolution solution = bestSolutions.getBestSolution(param,
    getSize(pjp.getArgs()));
    BaseMethodsGenerator.MethodInstance methodInstance =
    methods.get(solution.getNumberMethod());
    Object[] methodParams =
    representationGenerator.changeRepresentation(pjp.getArgs(),
    solution.getParam());
    return methodInstance.invoke(methodParams);
}
```

### 4.2.3 Summary

For adaptation of legacy code of matrix multiplication with AOP we implemented extension to base architecture which was described in previous chapter (3). The designed architecture is flexible which allows changing context parameters using two implemented methods of `BaseParameterGenerator`. We considered five algorithm implementation variants for matrix multiplication: `Baseline, Inline, BaselineSparse, Recursive and Strassen`; and two data structure representation variants: `dense` and
We implemented offline learning with AOP which uses BaseAspect to change data structure representation and choose best variant for current context. The use of the dispatch table has some benefits, e.g., getting best data structure representation and algorithm implementation variant is quite fast. The aspect implementation was used in both learning and testing phases.

4.3 Graph algorithms

For graphs implementation we used two algorithms: recursive Depth First Search (DFS) algorithm. Algorithm has 8 different implementation variants and they are developed by students and teachers in work [6] (Fig. 4.12).

All these algorithms can be used with different data structure representations. In our case we considered five types of data structures: cyclic graph that consists of a single cycle; acyclic graph that does not contain cycles; random graph can contain cycles, loops; disconnected graph that contains nodes which are not connected to each other; and two parts graph that contains two random separated graphs.

The context parameters are size (the number of nodes) of graph and additional parameters are five data representation variants.

4.3.1 Profiling and learning with AOP

For learning with AOP GraphInputValueGenerator class was implemented, and for context parameters generation GraphParameterGenerator class was implemented. In order to compare with implementation of matrix multiplication graph algorithms implementation variants recursively invoke another inner method that cannot be used in aspect around advice. In this case, in learning phase the aspect advice does not use the best fit data representation for current context parameters while there is no recursion and, therefore, it cannot use the results from the previous learning iterations. But from the other side, we have five different data representation variants and eight different algorithm implementation variants for DFS which can be used in learning phase. As for MM the method execution time is measured and the best-fit data representation with algorithm implementation variant are defined and saved. Additionally we implemented the conversion of data structure representation with methods that generate a cyclic, acyclic, random, disconnected and two parts graphs.

The code snippet bellow shows the recursive DFS algorithm that has one input parameter, which is graph type:

```java
(1)@GraphMethod(number = 0, isParallel = false)
(2)public List<Node<E>> dfs(DirectedGraph<E> graph){
(3)LHashSet<Node<E>> visitedNodes = new LHashSet<Node<E>>(){};
(4)    for(Iterator<Node<E>> it = graph.iterator(); it.hasNext();){
(5)        Node<E> n = it.next();
(6)        visitedNodes = nodeDFS(n, visitedNodes);
```
Each algorithm implementation variant for GA has annotation, cf. line 1, named `GraphMethod()` with two parameters: the number of algorithm implementation variant and boolean value that defines the parallel or sequence algorithm execution. We specified the package name “algorithms.GraphAlgorithms” in `GraphMethodGenerator`. Eight algorithm implementation variants are automatically collected. The code snippet above represents the generation of one input method value of algorithm implementation variant for each context parameters. To generate context parameters, we implemented two methods in `GraphParameterGenerator` which are represented in the code snippet below:

```java
@Override
(1) protected List<Parameter> generateParameters() {
(2) List<Parameter> parameterList = new ArrayList<Parameter>();
(3) parameterList.add(new Parameter(ParameterType.FIRST_DENSE, new Object[]{true}));
(4) parameterList.add(new Parameter(ParameterType.PARALLEL, new Object[]{false}));
(6) return parameterList;
}

@Override
(5) protected List<Parameter> generateAdditionalParameter() {
(6) List<Parameter> parameterList = new ArrayList<Parameter>();
(7) parameterList.add(new Parameter(ParameterType.GRAPH_TYPE, new Object[]{
{GraphType.random,
(8) GraphType.small_cyclic, GraphType.small_acyclic, GraphType.two_parts, GraphType.disconnected}
}})
(9) return parameterList;
}
```

We consider two parameter types for GA: graph in dense representation (cf. line 3) (density defines how many edges per nodes in graph are generated), and parallel or none-parallel algorithm implementation variant choice (cf. line 4). We do not have parallel algorithm implementation variant of DFS, so we specified false value for the PARALLEL parameter. In the method generated `AdditionalParameters()` we specified five possible data structure representation for graph algorithm (cf. line 7-8): cyclic, acyclic, random, disconnected, two parts.

### 4.3.2 Composition block with AOP

In this sub-section we describe the implementation of components in the Composition block for GA with AOP and provide code examples. Notice, that the same implementation as for MM was used:

```java
@Around("@annotation(LogAction) && (@withincode(LogAction)) || @withincode(ThreadAnnotation))")
public Object training(ProceedingJoinPoint pjp) throws
IllegalArgumentException, InvocationTargetException, IllegalAccessException {
(3) Map<Integer, BaseMethodsGenerator.MethodInstance> methods =
methodsGenerator.getAllMethods(param);
(4) BestSolution solution = bestSolutions.getBestSolution(param,
getSize(pjp.getArgs()));
(5) BaseMethodsGenerator.MethodInstance methodInstance =
methods.get(solution.getNumberMethod());
```
Object[] methodParams = representationGenerator.changeRepresentation(pjp.getArgs(), solution.getParam());

return methodInstance.invoke(methodParams);

There are only two differences from MM: the first one is that aspect is not executed during the learning phase because the algorithm implementation variant does not have support for recursion; the second is that graph algorithms do not have parallel implementation variants.

4.3.3 Summary
For adaptation of legacy code of graph algorithms with AOP we implemented extension to base architecture which was described in chapter 3. The designed architecture is flexible which allows changing context parameters using two implemented methods of BaseParameterGenerator. We considered eight algorithm implementation variants for depth-first search (DFS) graph algorithm: alev, lotus, players, steamp, thinkers, ukraine, xz, zeroday; and five data structure representation variants: cyclic, acyclic, random, two_parts – graph consisting with two graphs and disconnected graph. We implemented offline learning with AOP which uses BaseAspect to change data structure representation and to choose best variant for current context. The aspect implementation was used in testing phases.

4.4 Conclusions
The extension of Template pattern was implemented and described in Chapter 3. This allows adaptation of the legacy codes to CAC with AOP approach with possibility of changes in data representation. Additionally, the classes were implemented in order to provide data conversion without additional efforts to reengineer or redesign existing code. Two applications were considered in our implementation. These are Matrix Multiplication and Graph algorithms. The implementation of the Composition block that was used for both applications is the same. In the next chapter we evaluate our implementation in order to provide the evidence to our approach and achieve the thesis goals.
5 Experiments

In our experiments we considered two application domains that were described in the previous section: Matrix Multiplication (MM) problem and depth-first search (DFS) Graph algorithms.

Experiment shows the feasibility and usefulness of the AOP approach in CAC adaptation for improving performance in terms of method execution and conversion representation time.

5.1 Experimental set-up

All experiments are executed on two different multi-core machines with native JVM virtual machine parameters: (M1) a 2 core Samsung PC running Ubuntu (12.10) on an Intel Dual Core i5 M560 at 2.67GHz and 4GB RAM, and (M2) 4 core Macbook Pro (OS X 10.8.2) on 2.3 GHz Intel Core i7 and 8 GB RAM.

5.2 Matrix multiplication algorithms

According to implementation of the Composition block with AOP context parameters such as size of matrixes, density that that indicates how many non-zero elements in matrix, and data structure representation variants were generated. The matrix size $N$ was chosen in interval from 1 to 272 with stride of 32. Data structure representation variants had two possible variants dense and sparse. Number of available processors $P$, $p=0$ indicates that non-parallel algorithm implementation variants of MM are used and $p=1$ the parallel algorithm implementation variants are used in learning and testing phase. We used five algorithm implementation variants for MM: (1) Inline – the standard implementation with the nested loops; (2) Baseline – reducing a matrix multiplication to a number of matrix-vector multiplications and then each of such, in turn to a number of vector multiplications; (3) BaselineSparse – a variation of Baseline optimized for sparse matrices; (4) Recursive – an algorithm that reduces matrix multiplications to eight multiplications of eight sub-matrices; and (5) Strassen – an algorithm that uses recursion with seven multiplications, this algorithms uses threads.

For the MM we had 5 algorithm implementation variations, where three of which use recursion and one includes threads if the $p=1$. We had four conversion data representation variants such as converting $m1$ to certain data structure(dense or sparse), converting $m2$ to certain data structure(dense or sparse), converting both of them or none. In our approach we used Dispatch Table for saving and retrieving best algorithm implementation variant and data structure representation. During the execution of Profiling and Learning blocks for all algorithm implementation variants with all variants of data structure representation the execution time was calculated. Execution time was measured as average execution time of data representation change and algorithm implementation variant execution for 256 iterations. The best execution time that defined the best implementation variant with data representation was saved to the dispatch table.

We conducted experiment using conversion to the specific data representation with Aspect approach and without it as Manual approach in order to analyse the performance as shown on Fig. 5.1-5.2. Manual approach is approach that adapt legacy code manually by reengineering CAC for different applications. We did experiment with Manual approach where the best implementation variant was retrieved from dispatch table on each iteration in order to compare the results with our aspect-oriented approach. We considered comparing Baseline algorithm implementation variant for MM to Aspect and Manual approaches because it is rather fast and non-parallel.
Therefore, we compare Aspect approach with data structure representation to Aspect without data structure representation.

Figure 5.1 Execution time of Baseline and CAC approaches for Matrix Multiplication (M1)

In Fig. 5.1 the AOP approach with data structure representation conversion increases performance on the larger problem size in comparison with Baseline and Aspect or Manual without representation algorithm implementation variant. The same experiment was conducted for second machine (M2) and shown in Fig. 5.2.

Figure 5.2 Execution time of Baseline and CAC approaches for Matrix Multiplication (M2)

Experiment results from both machines M1 and M2 are represented in Table 5.1

<table>
<thead>
<tr>
<th>Platform</th>
<th>Problem size</th>
<th>Baseline[ms]</th>
<th>Aspect[ms] without representation</th>
<th>Manual[ms] without representation</th>
<th>Aspect[ms] with representation</th>
<th>Manual[ms] with representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC 2 cores</td>
<td>100</td>
<td>34</td>
<td>40</td>
<td>54</td>
<td>24</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>250</td>
<td>493</td>
<td>389</td>
<td>400</td>
<td>230</td>
<td>263</td>
</tr>
</tbody>
</table>

26
5.3 Graph algorithms

We conducted two experiments on Graph algorithms, one where we altered the algorithm without AOP approach (Manual) and a second one with AOP approach (Aspect). We selected between eight different depth-first search (DFS) algorithm implementation variants that were implemented by seven groups of students, which we distinguish with different names: alev, lotus, players, steam, thinkers, ukraine, xz, zeroday. They all have common interface with different implementations that satisfied the prerequisites for adaptation to CAC.

Each experiment was divided into two parts: learning and testing. For learning, we generated graph nodes from 1 to 1000 in steps 50 (size). For each size (the number of nodes) we generated graph with random edges between nodes. We considered five data representation variants for graphs algorithm. These are: cyclic, acyclic, random, two_parts – graph consisting of two graphs and disconnected graph. During the testing phase, we measured the execution time for graphs with number of nodes from 1 to 1000 as shown in Fig. 5.3.

![Figure 5.3 Execution time of Alev and CAC approaches for Graph algorithms (M1)](image)

The results presented on Fig. 5.3 show that the AOP approach (Aspect) with data structure representation conversion increase performance on the larger problem size comparing to alev algorithm implementation variant. We considered alev algorithm for comparing to our approach because it is rather fast. The same experiment was conducted for second machine (M2) and is shown in Fig. 5.4.

<table>
<thead>
<tr>
<th>(M1)</th>
<th>500</th>
<th>230342</th>
<th>9453</th>
<th>10345</th>
<th>468</th>
<th>420</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC 4</td>
<td>100</td>
<td>27</td>
<td>29</td>
<td>32</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>cores</td>
<td>250</td>
<td>320</td>
<td>289</td>
<td>300</td>
<td>192</td>
<td>168</td>
</tr>
<tr>
<td>(M2)</td>
<td>500</td>
<td>12424,5</td>
<td>7650</td>
<td>7988</td>
<td>361</td>
<td>334</td>
</tr>
</tbody>
</table>

Table 5.1 Speed-up of different CAC approaches for MM with and without data representations.
Experiment results from both machine M1 and M2 are presented in Table 5.2.

<table>
<thead>
<tr>
<th>Platform</th>
<th>Problem size</th>
<th>DFS(alev)[ms]</th>
<th>Aspect[ms]</th>
<th>Manual[ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC 2 cores (M1)</td>
<td>200</td>
<td>5</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>600</td>
<td>25</td>
<td>17</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>1000</td>
<td>79</td>
<td>43</td>
<td>37</td>
</tr>
<tr>
<td>PC 4 cores (M2)</td>
<td>200</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>600</td>
<td>23</td>
<td>10</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>1000</td>
<td>59</td>
<td>29</td>
<td>23</td>
</tr>
</tbody>
</table>

Table 5.2 Speed-up of different CAC approaches for Graph Algorithm (in msec).

5.4 Summary

We conducted two experiments for Matrix Multiplication problem and Graph algorithms. All results are presented in the tables and figures obtained from experiments on two machines with different technical characteristics. The Manual approach has still better results than Aspect approach hence the time that is spent on the reengineering or redesign (with Manual approach) is saved and differences between execution time values are not significant comparing to Manual approach.

On M1, the extension for AOP based approach in MM with data representation has an average speed up of 493msec/230msec = 2.14 over sequential Baseline for problem size 250. On M2, the extension of AOP based approach in MM with data representation has an average speed up of 1.6 over sequential Baseline for problem size 250. Manual approach has a bit higher speed up that is 1.3 than Aspect approach.

On M1, the extension of AOP based approach in GA with data representation has an average speed up of 1.8 over sequential and faster Alev algorithm for problem size 600. On M2, the extension AOP based approach in GA with data representation has an average speed up of 2.0 over sequential Alev for problem size 600. Manual approach has a bit higher speed up that is between 1.0 and 1.05, 1.0 and 1.3 for M1 and M2 respectively.

For matrix multiplication, the Manual approach introduces the average overhead 10.2% and 7.4% over the extended AOP based approach for machines M1 and M2 correspondently. For graph algorithms the overhead is smaller: on average about 6% and 20%, for M1 and M2 respectively.
5.5 Lines of Code

We used the line of code $LOC$ metrics in order to measure the effort required for adaptation for Manual approach $LOC_m$ and Aspect approach $LOC_a$. We calculate the programming effort with the metrics $PI=LOC_m/LOC_a$. For matrix multiplication problem, $LOC_m = 589$ and $LOC_a = 175$, $PI= 3.3$, for graph algorithms problem $LOC_m = 341$ and $LOC_a = 120$, $PI=2.8$.

The experiments showed that our extension to base approach can effectively adapt existing legacy codes to CAC which makes it possible for them to run efficiently even on multicore machines. Hence the speed-ups of applications with CAC adapted manually is slightly higher than with the AOP-based approach, the performance overhead is quite small. Moreover, AOP-based adaptation requires a smaller programming effort as the line of code is 3 times less than Manual approach.
6 Related work

The aspect-oriented approach is one of the solutions for the developing the adaptable software in such domains as services[16], mobile[15], multi-agent systems [17,16]. Furthermore the object-oriented software applications with aspect-oriented mechanism allows supporting context-awareness in terms of dynamically adapting the application behaviour by handling the current context [19]. For instance Li et al. [5], proposed approach that based on aspect-oriented programming to perform context-aware service composition on the fly. They implemented context weaver algorithm that allows Web services to be composed as the context changes. Tanter et al. [7], developed an open framework for context-aware aspects that supports the definition of context awareness constructs for aspects, including the ability to refer to past contexts, and to provide domain and application-specific constructs. In work [9] they provided safe, efficient, and highly customizable Policy-based design without profiling and composition possibilities for adaptation legacy code. Another approach proposed in [10] used well-known library generators as ATLAS\(^2\), FFTW\(^3\), and SPIRAL\(^4\) to define the sorting algorithms with best performance. They used the Winnow algorithm for learning stage and implemented runtime adaptation mechanism for improving performance only for one domain and does not support others as matrix operations, etc.. JPed[11] is an extension to Java for supporting predicate dispatch that allows determining the method implementation to be invoked upon the specifying the conditions under which the method should be invoked.

The specialization of code is one of the approach for optimization and adaptation the software applications. In work [12] approach that allows recognition of object-oriented programs by specialized methods with respect to the methods input values during the compilation time or at runtime is proposed. Hence, this approach requires additional programming efforts for each method that should be specialized. Another work [14] proposed the adaptive algorithm selection framework that provides offline and online selection of the best algorithms, however it does not considers data representation. The work mostly related to ours is [8], where the designed library chooses the best data representation with conversion mechanism and best algorithm for matrix multiplication. However, opposite to our work, this approach required additional efforts to redesign or reengineer the library for another application domains as for example Graph algorithms.

\(^2\)http://math-atlas.sourceforge.net/
\(^3\)http://www.fftw.org/
\(^4\)http://www.spiral.net/
7 Conclusions and future work

Context-aware composition approach allows improving application performance if the design of application corresponds to prerequisites for adaptation that were described in chapter 2. Hence this approach requires additional programming efforts to reuse it with another application domains. To avoid this, in our work we used aspect-oriented adaptation approach that allows improving performance of applications and, therefore, adding new applications without additional efforts to change design. This approach saves development time for legacy code adaptation. We reviewed online and offline learning strategies, strategies that can be applied to self-adaptive systems. Online approach requires a major expert that can be represented by a set of offline-learned decision functions, hence along with all benefits, the major drawback of online learning is that it will add to the workload on a live system. Offline approach can require learning time ranging from few seconds to several hours as well as explicit training data. The basic learning requirement for self-adaptive systems is that the learning process must be fast enough, dynamic and continuous. Hence, in this work we considered a context-aware composition, where the learning time was not taken into account. We used offline approach in order to provide learning with already known training data which consists of context information. The main benefit of using offline learning with respect to our aspect-based approach is knowledge from previous learning iteration that could be used in current learning phase.

Aspect-oriented approach allows adapting legacy code and reusing CAC components for different application domains. Developers can change or add new aspects to CAC rather than implement new CAC patterns with data representation conversion. We considered offline learning with around aspect advice that allows selecting and executing the best algorithm and moreover change data representation to the best one. We extended the design of CAC adaptation that was presented in [1]. This extension allows adapting different applications in terms of changes in data representation variants. In order to test and evaluate our approach we implemented Matrix Multiplication problem and Graph algorithms with conversion data representation. The Profiling block was extended by generator for all possible data representation variants, BaseParameterGenerator, and profiling all data representation with changing to another one by Representation class during the learning phase. In the Composition block we implemented aspect that was able to change representation for both Matrix Multiplication and Graphs algorithm problems.

7.1 Project evaluation summary

The implemented extension was evaluated on Matrix Multiplication and Graph Algorithm applications to compare performance with the manual adaptation approach. The experiments results with changing data representation variants showed slightly lower speed comparing to manual approach. However, the development efforts of AOP approach for both applications are on average 3 times less comparing to manual adaptation approach($LOC_a = 589$ for MM and $LOC_a = 341$ for GA; $LOC_m = 175$ and $LOC_a = 120$ for MM and GA respectively). We improve application performance compare with Aspect approach without data representation. Hence, performance of Manual approach is slightly better than Aspect approach: 7% for MM and 6% for GA.

7.2 Future work

As for future work, the data representation changing can be improved by changing specific current piece of data for current context, for example, conversion of submatrix
instead of whole matrix in Matrix Multiplication algorithm. Moreover, the online learning with data representation conversion can be implemented.
8 References


