ABCLibScript: A Directive to Support Specification of An Auto-tuning Facility for Numerical Software [*]

Takahiro Kagiri, Kenji Kise, Hiroki Honda and Toshitsugu Yuba
Graduate School of Information Systems,
The University of Electro-Communications
/ Japan Science and Technology Agency, PRESTO
Phone: +81-424-43-5642 FAX: +81-424-43-5644
E-mail: katagiri@is.uec.ac.jp

Abstract

In this paper, we describe the design and implementation of ABCLibScript, which is a directive that supports the addition of an auto-tuning facility based on the FIBER framework. ABCLibScript limits the function of auto-tuning to numerical computations. For example, the block length adjustment for blocked algorithms, loop unrolling depth adjustment and algorithm selection are crucial functions in ABCLibScript. To establish these three particular functions, we make three kinds of instruction operators, *variable*, *unroll*, and *select*, respectively. We focus on the reduction and support of development for auto-tuning software to innovate the limited functions of ABCLibScript. As a result of our performance evaluation, we showed that a non-expert user obtained a maximum speedup of 4.3 times by applying ABCLibScript to a program compared to a program without ABCLibScript.

Keywords: ABCLibScript, Directive, Software Auto-Tuning, FIBER, Pre-processor

1 Introduction

Recently, many numerical libraries with an "auto-tuning facility" have been developed, e.g., PHiPAC [2], ATLAS [1], FFTW [6]. We refer to a library with an auto-tuning facility as *SATF (Software with Auto-Tuning Facility)*. The early research on these libraries established the effectiveness of SATF [11].

Although SATF is effective from the viewpoint of performance, we are currently developing a "specialized" auto-tuning facility. For example, direct solvers, iterative solvers, dense solvers, and sparse solvers are included in numerical software packages, but we have no way to adapt a single auto-tuning facility to all of these solvers. For this reason, we have developed a directive,

[*]This paper was submitted to IS Technical Report UEC-IS-2004-7 in the Graduate School of Information Systems, the University of Electro-Communications in September 30th of 2004.

named ABCLibScript, which can support the addition of an auto-tuning facility. The auto-tuning facility is restricted to the function of numerical processing to reduce the complexity of the ABCLibScript implementation.

The organization of this paper is as follows. Section 2 explains the FIBER framework, which is the base technology of the auto-tuning used in ABCLibScript. In Section 3, the development policy for ABCLibScript is described. Section 4 describes the implementation details of ABCLibScript, especially the ABCLibScript API (application programming interface). Section 5 provides some programming examples using ABCLibScript. Finally, Section 6 summarizes the observations of this study.

2 Outline of Auto-tuning with the FIBER Framework

FIBER, which is the framework of auto-tuning, was created to establish wide applicability of SATFs [16, 15, 9]. For instance, FIBER allows us to increase the number of adaptable software packages, and achieves high accuracy for estimated parameters. To establish these goals, FIBER has three kinds of timing for parameter optimization: *(1) IOL(Install-time Optimization Layer)*, which is performed when the software is installed, *(2) BEOL(Before Execute-time Optimization Layer)*, which is performed when the end-users fix the special parameters defined by the software developers (e.g., the problem size to execute), and *(3) ROL(Run-time Optimization Layer)*, which is performed when the target processes run. The software framework to establish this approach is called *FIBER (Framework of Install-time, Before Execute-time, and Run-time optimization layers.)*

2.1 Two Kinds of Users and Software Organization

The main target process of FIBER is numerical processing, thus the users are defined as follows:

- Software developer
- Software user (end-user)

The software developer is a user who adds the auto-tuning facility to their own software by using the instruction operations of ABCLibScript, which are provided by the FIBER toolkit developer. On the other hand, the software user (end-user) is a user who uses the software with the auto-tuning facility which was developed by the software developer.

Figure 1 shows the process flow in FIBER from the viewpoints of the above two kinds of users. Figure 1 indicates that the process flow is different for the two users. We explain the usage scenario for FIBER in the following.

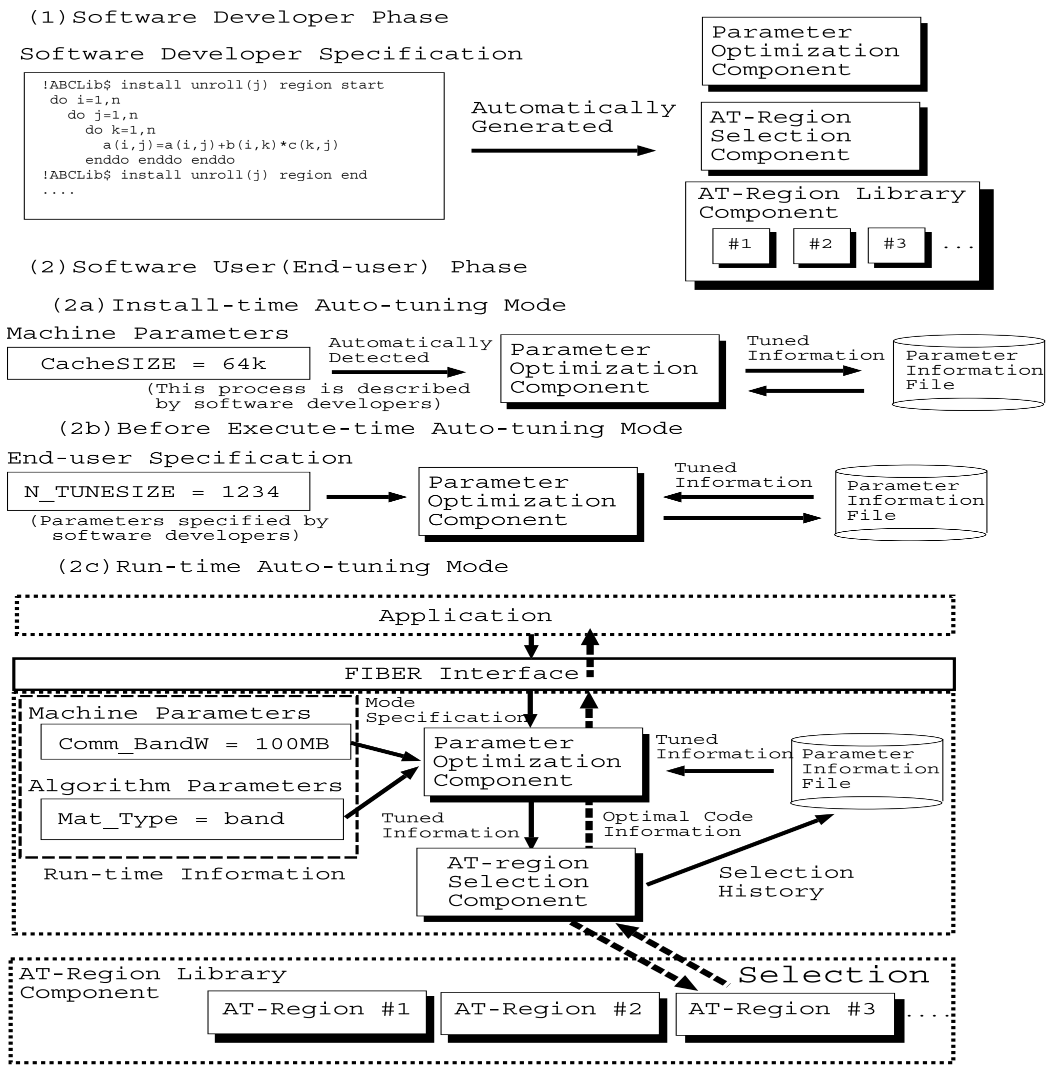
(1)Software Developer Phase
Software Developer Specification
!ABCLib$ install unroll(j) region start
do i=1,n
do j=1,n
do k=1,n
a(i,j)=a(i,j)+b(i,k)*c(k,j)
enddo enddo enddo
!ABCLib$ install unroll(j) region end
Automatically Generated
Parameter Optimization Component
AT-Region Selection Component
AT-Region Library Component
#1
#2
#3
(2)Software User(End-user) Phase
(2a)Install-time Auto-tuning Mode
Machine Parameters
CacheSIZE = 64k
Automatically Detected
(This process is described by software developers)
Parameter Optimization Component
Tuned Information
Parameter Information File
(2b)Before Execute-time Auto-tuning Mode
End-user Specification
N_TUNESIZE = 1234
(Parameters specified by software developers)
Parameter Optimization Component
Tuned Information
Parameter Information File
(2c)Run-time Auto-tuning Mode
Application
FIBER Interface
Machine Parameters
Comm_BandW = 100MB
Algorithm Parameters
Mat_Type = band
Run-time Information
Mode Specification
Parameter Optimization Component
Tuned Information
Parameter Information File
Tuned Information
Optimal Code Information
AT-region Selection Component
Selection History
AT-Region Library Component
AT-Region #1
AT-Region #2
AT-Region #3
Selection

Figure 1: Process flow in FIBER.

- *Software Developer Phase*:
  Software developers perform the following processes to generate the auto-tuning facility in the FIBER framework.

  (1a) The software developer, who wants to add the auto-tuning facility to their software, writes the operations in their source programs using the language provided by the FIBER toolkit developer for the target regions, i.e., computation kernels, in the numerical computation routines. We call this region an *AT region*.

  (1b) The software developer applies a pre-process tool which is provided by the FIBER toolkit developer. The pre-process tool generates new source codes with the auto-tuning facility. The automatically generated auto-tuning facility contains three components: (i) Parameter Optimization Component, (ii) AT-Region Selection Component, and (iii) AT-Region Library Component. These components are explained in the section below.

(1c) The software developer opens the source codes to the end-users as a library. Or, the software developer compiles the source codes with a compiler, and the compiled object code is also opened to the end-users.

- *Software User (End-user) Phase*:
  The end-user uses the auto-tuning software opened by the software developer. The following three optimization modes are performed to obtain high performance using the three kinds of timing.

  (2a) *Install-time Optimization Mode:* This mode is performed when the software is installed into the end-user's environment. The install-time optimizer tries to find the hardware parameters, e.g., cache sizes and message communication latency, to establish efficient implementations corresponding to the end-user's machine environments. The end-user is unaware of this optimization since the optimization can be totally hidden in the installation or compilation processes for the target software.

  (2b) *Before Execute-time Optimization Mode:* This mode is performed after special parameters are fixed, based on the end-user's knowledge. For example, the problem size to execute their software is a special parameter because the system cannot know this value in advance. By using the fixed parameter information, optimization is performed in this mode. This optimization also includes the selection of the best algorithm. Compilers cannot perform such an algorithm selection, because of the lack of knowledge of the end-user's algorithms.

  (2c) *Run-time Optimization Mode:* This mode is performed when the target auto-tuning regions are executed or called. The run-time optimizer tries to obtain run-time information, e.g., input data characteristics such as the band matrix for numerical computation, the load of the machines, and the current communication performance (such as the communication bandwidth for a GRID environment.) Since the run-time optimization optimizes the target regions with "dynamic" information, it can set the best parameter which cannot be optimized in the first two optimization modes.

2.2 Components

The FIBER framework supports the following components:

- **Parameter Optimization Component:** This component optimizes developer-specified library parameters in the PTL (Parameter Tuning Layer.) There are three different kinds of timing for the optimizations (install-time, before execute-time, and run-time.)

- **AT-Region Selection Component:** This component selects the best auto-tuning region, taking into account information from the parameter information file and the run-time optimizer.

- **AT-Region Library Component:** This component is automatically generated according to the pre-process specifications from the software developer. The commands of this pre-process are provided by the FIBER toolkit developer. The regions of auto-tuning (AT-Regions) are specified by the software developer through ABCLibScript. According to the software developer's instructions, the specified AT-regions are separated and formed into several components. The aggregation of these components forms the *AT-Region Library.*

3 Design of ABCLibScript

3.1 Design Policy

The design of ABCLibScript is based on the following four design policies.

1. Easy Specification: The specification is dedicated for numerical processing. Hence, the provided functions are limited.

2. Non-disturbance for Original Program Execution: The software developer specifies the operations using the directives in the original program. Hence, the original program execution is not disturbed.

3. Generation of High-readability Codes: The pre-processor provided can generate high-readability codes according to the directive specified by the software developer.

4. Simplification of the Auto-tuning Process: Using the concept of two kinds of system parameters, which are PP (Performance Parameter) and BP (Basic Parameter), the auto-tuning process is simplified.

ABCLibScript has the following limited instruction operators to dedicate to the numerical processing.

- `Unroll`: Loop unrolling depth adjustment to loop unrolled codes.

- `Variable`: Blocking length adjustment to blocked algorithms.

- `Select`: Algorithm selection based on the user's knowledge.

The ABCLibScript toolkit developer supplies a pre-processor, which can interpret the directives specified by the software developer. In addition, it can generate codes which the software developer can easily read, because the code is written in the computer language used by the software developer. For this reason, the software developer can manage the auto-generated codes. This approach also has another advantage. Since the auto-generated codes have high-readability, the software developer can find and understand the auto-tuning process. The policy of ABCLibScript, hence, is based on a while box approach.

ABCLibScript defines the auto-tuning according to the concept of two kinds of system parameters, PP and BP. The system parameters of PP are determined to minimize the *Cost Definition Function* F, which is defined as the cost in the target AT region when the system parameters of BP are fixed (See [16, 15, 9].) We can also say that ABCLibScript is a directive which can specify the auto-tuning facility to fix the system parameters PP and BP defined by the software developer. By using this concept, specification makes the auto-tuning process simple.

3.2 Specification Format

3.2.1 Specification of Performance Parameters for the Software Developer

We have explained that the software developer should define the performance parameters (PPs), the basic parameters (BPs), and auto-tuning regions (AT regions) in the source program* .

The end-user can define the basic parameter BP in the Before Execute-time Optimization. We also can say that the auto-tuning in FIBER is a process that estimates the best PP parameter automatically, based on a BP fixed by the software developer or end-user, or both.

An overview of the specification is listed in Figure 2. The line `!ABCLib$` specifies ABCLibScript for auto-tuning in the source program.

```
!ABCLib$ ⟨Auto-tuning Type⟩ ⟨Function Name⟩ (Target Variables) region start
!ABCLib$ ⟨Details of Function⟩sub region start
                AT Region
!ABCLib$ ⟨Details of Function⟩sub region end
!ABCLib$ ⟨Auto-tuning Type⟩ ⟨Function Name⟩ (Target Variables) region end
```

Figure 2: Auto-tuning Specification Format in ABCLibScript.

The ⟨Auto-tuning Type⟩ and ⟨Function Name⟩ in Figure 2 are called *instruction operators*.

* For the basic parameter BP, the variable n for program size is a default parameter in this system. If you want to add the basic parameter, `ABCLib_BPset`, which is an API for the software developer, can be used in the ABCLibScript specification. The explanation of this API is shown later in this paper.

The instruction operator ⟨Auto-tung Type⟩ can specify the three kinds of timing—install-time optimization (`install`), before execute-time optimization (`static`), and run-time optimization (`dynamic`). The way to process the target code and the details for the auto-tuning method can be specified by the instruction operator ⟨Function Name⟩.

3.2.2 Instruction Operators and Co-operators

The instruction operators are listed in Figure 3, and the instruction co-operators are listed in Figure 4.

- ⟨Auto-tuning Type⟩ ::= (`install` | `static` | `dynamic` | ⟨Formula⟩)

  `install` : Specifying Install-time Auto-tuning.

  `static` : Specifying Before Execute-time Auto-tuning.

  `dynamic` :Specifying Run-time Auto-tuning.

  ⟨Formula⟩ ::= Formulas according to computer language grammar
  for target code.
- ⟨Function Name⟩ ::= (`define` | `variable` | `select` | `unroll`)

  `define` : Specify definition of parameters.

  `variable` : Specify variable parameters.

  `select` : Specify selection among AT regions.

  `unroll` : Specify loop unrolling.

Figure 3: Instruction operations in ABCLibScript

3.3 Target Computer Language

By nature, the ABCLibScript specification does not depend on the computer language which the software developer used to write the program code. However, for implementation of the pre-processor of the ABCLibScript directives, a target language should be chosen.

The numerical software is the target application in this directive. Fortran90 and MPI-1 (Message Passing Interface-1), hence, are used in the target application program. We state the ABCLibScript specification, which takes into account the target computer language, as $ABCLibScript(Fortran90, MPI-1)$.

3.4 How To Use the Pre-processor

The FIBER toolkit developer supplies a pre-processor, which can interpret the directive by ABCLibScript and generate codes with the auto-tuning facility. The name of the pre-processor is `ABCLibCodeGen`.

- `name` ⟨Characters⟩ : Describe the name of AT region. (Available in all functions)
- `parameter` (⟨Property Specification⟩ ⟨Variable Name⟩, [⟨ Property Specification⟩ ⟨Variable Name⟩, ...])
  : Specify output variables to parameter information file, or input variables from parameter information file.
  Or, declaring basic parameters.
  ⟨Property Specification⟩ ::= [`in` | `out` | `bp`]
  `in` : input parameters, which are defined in outer part and referred in this AT Region.
  `out` : output parameters, which are defined in this AT region.
  `bp` : Basic Parameters. (Available in all functions)
- `select sub region`(`start` | `end`) : Specify selection process. (when specifying the select operator)
- `according` (⟨Condition Formulas⟩ | `estimated` ⟨Formulas⟩) : Select the AT region based on the following conditions.
  ⟨Condition Formulas⟩ ::= [(`min` ⟨Variable Name⟩ | `condition` ⟨Conditions⟩) ⟨Connection Operator⟩]
  ⟨Connection Operator⟩ ::= [`.and.` | `.or.`] ⟨Condition Formulas ⟩
  `estimated` ⟨Formulas⟩: Select the best process based on the cost formula defined by software developers.
  (when specifying select operator)
- `varied (⟨ Variable Name ⟩ [ ,⟨ Variable Name ⟩,... ] ) from X to Y` : Specify the range of variable (from X to Y.)
  (when specifying the functions of variable and unroll)
- `fitting` ⟨Method⟩ `sampled` ⟨Range⟩ : Specify the method to estimate parameters.
  ⟨Method⟩ ::= [`least-squares` ⟨Order⟩ | `user-defined` ⟨Formulas⟩ | `auto`]
  `least-squares` : Use the least-squares method with linear polynomial formula.
  the order is defined in ⟨Order⟩.
  `user-defined` : Use the least-squares method with the software developer defined formula.
  `auto` : Determine the parameter estimation method by the system.
  ⟨range⟩ ::= [⟨Values⟩ | `auto`] : Specify the regions to use parameter estimation.
  If you specify ⟨Method⟩ as `auto`, you can omit this.
  ⟨Values⟩: Specify values for parameters.
  `auto` : Determine sampling range automatically, when you do not specify the operator of `fitting`.
  The optimal values are specified in which all regions defined the operator of varied are searched.
  (when specifying functions of `variable` and `unroll`)
- `pripro sub region` (`start` | `end`): Describe pre-process before the AT region is called. (Available in all functions)
- `postpro sub region` (`start` | `end`): Describe post-process after the AT region is called. (Available in all functions)
- `debug` (⟨Variables⟩,[⟨Variables⟩]) : Specify debug variables when the AT region runs. (Available in all functions)
  ⟨Variables⟩ ::= [`bp` | `pp` | ⟨Arbitrary Values⟩]
  `bp` : print information of basic parameters; `pp` : print information of performance parameters;
  ⟨Arbitrary Values⟩ : Print information for the described variables;

Figure 4: Instruction co-operations of ABCLibScript.

The following command is performed to generate a parallel numerical computation program with the auto-tuning facility in the program (test.f) based on the specification of $ABCLibScript(Fortran90, MPI-1)$.

```
>ABCLibCodeGen test.f
```

The following run-time options can be specified in the pre-processor of `ABCLibCodeGen`.

[Run-time option] `-debug`

`OFF`: Do not generate debugging codes. (Default)

`ON`: Generate debugging codes with level x, which can be specified by the system variable `ABCLib_DEBUG`.

[Run-time option] `-visualization`

`OFF`: Do not output to an auto-tuning log file (Default).

`ON`: Output to an auto-tuning log file

[Example] `> ABCLibCodeGen -debug ON -visualization ON test.f`

Generate the codes with the auto-tuning facility in the program test.f. The generated codes contain a debugging code and a routine to output an auto-tuning log file. By using a visualizer, which is now being developed, the process of auto-tuning will be visualized with the output log file.

4 Implementation of ABCLibScript

4.1 ABCLibScript APIs

ABCLibScript has APIs to support the detailed specification of auto-tuning with directives. The main goal of the APIs is to support detailed software developer descriptions, but end-users do not use all the functions of the APIs[†] .

To execute auto-tuning, the following API is specified.

[API 1] `ABCLib_ATexec ( ABCLib_ATkinds, ABCLib_ATroutines )`

The routine of `ABCLib_ATexec` executes the auto-tuning specified by `ABCLib_ATkind` to the target regions specified by `ABCLib_ATroutines`. The argument of `ABCLib_ATkinds` specifies the type of auto-tuning. The following four kinds of constant values, which are defined in the header file `ABCLibScript.h`, can be specified.

- `ABCLib_INSTALL`: Specify the Install-time Auto-tuning;
- `ABCLib_STATIC`: Specify the Before Execute-time Auto-tuning;
- `ABCLib_DYNAMIC`: Specify the Run-time Auto-tuning;
- `ABCLib_ALL`: Specify all auto-tuning;

The argument of `ABCLib_ATroutines` specifies the target AT regions. This argument is specified with the arbitrary AT regions, which are named by the software developer, or with the variables `ABCLib_ATname` defined in the header file `ABCLibScript.h`. The common variables are shown as follows:

- `ABCLib_AllRoutines`: For all routines;

[†] Basically, an end-user does not want to use the API functions because they only want to use the software opened by the software developer. The end-user should only use the API when performing a before execute-time optimization in FIBER.

- `ABCLib_InstllRoutines`: For Install-time Auto-tuning;
- `ABCLib_StaticRoutines`: For Before Execute-time Auto-tuning;
- `ABCLib_DynamicRoutines`: For Run-time Auto-tuning;

[Example] `!ABCLib$ call ABCLib_ATexec (ABCLib_INSTALL, ABCLib_InstallRoutines)` : Install-time auto-tuning is performed in all of the AT regions specified by the install-time auto-tuning.

[API 2] `ABCLib_ATset (ABCLib_ATkinds, ABCLib_ATroutines)`

The routine `ABCLib_ATset` sets the auto-tuning type described in `ABCLib_ATkind` to the AT regions specified in `ABCLib_ATroutines`.

[API 3] `ABCLib_ATdel (ABCLib_ATroutines, DelName)`

The routine of `ABCLib_ATdel` deletes the AT regions specified by `DelName` in `ABCLib_ATroutines`, which is a list of registered AT region names. The argument of `DelName` specifies the AT region name. The AT region name is also specified in the instruction co-operator of `name` in each AT region.

[Example] `!ABCLib$ call ABCLib_ATdel(ABCLib_InstallRoutines,"MyMatMul")` : The AT region named "MyMatMul" is deleted as a candidate of the install-time auto-tuning.

[API 4] `ABCLib_ATInstallInit ( ABCLib_InstallRoutines )`

The routine of `ABCLib_ATInstallInit` sets a flag for the execution of install-time auto-tuning as not-executed. The target AT region is specified in `ABCLib_InstallRoutines`.

[API 5] `ABCLib_BPset ( BPvalName )`

The routine of `ABCLib_BPset` sets a new basic parameter (BP) specified in the argument of `BPvalName`.

[Example] `!ABCLib$ call ABCLib_BPset( "nprocs" )`

: The variable of `nprocs` is set as a new basic parameter (BP).

[API 6] `ABCLib_BPsetName (Kind, BPvalName, Name)`

The routine of `ABCLib_BPsetName` sets a variable name, which is specified in the argument of `Name`. The target variable is a basic parameter specified in `BPvalName` for fixing the basic parameter BP when auto-tuning is performed. For the argument of `Kind`, the meaning is

- `Kind` ::= [`STARTTUNESIZE` | `ENDTUNESIZE` | `SAMPDIST`], where `STATTTUNESIZE` : Start point information of sampling for basic parameter of `BPvalName`; `ENDTUNESIZE`: End point information of sampling for basic parameter of `BPvalName`; `SAMPDIST`: Stride information of sampling for basic parameter of `BPvalName`;

[Example] `!ABCLib$ call ABCLib_BPsetName("STARTTUNESIZE","nprocs",`

```
!ABCLib$ &   "ABCLib_NprocsStartSize")
```

: The variable for the start sampling point in auto-tuning for the basic parameter of `nprocs` is named `ABCLib_NprocsStartSize`.

[API 7] `ABCLib_BPsetCDF (BPvalName, CDFKind)`

The routine of `ABCLib_BPsetCDF` sets the estimation method for the parameter values, except for the sampling points in the basic parameter specified in the argument of `BPvalName` for the method specified in the argument of `CDFKind`. The methods specified in the argument of `CDFKind` are

- `CDFKind` ::= [`least-squares` ⟨Order⟩ | `user-defined` ⟨Formula ⟩ | `auto`],

where `least-squares` ⟨Order⟩ specifies the least-squares method using a linear polynomial in which ⟨Order⟩ specifies the order of the linear polynomial; `user-defined` ⟨ Formula ⟩ specifies the least-squares method in which the formula is defined by the user; `auto` specifies an estimation method that the system selects automatically;

The default process is the same method defined in the target AT region. If the user does not specify the estimation method even in the AT region, a linear polynomial formula of the third order is selected.

[Example] `!ABCLib$ call ABCLib_BPsetCFD("nprocs", "least-squares 5")`

: The estimation method for the basic parameter `nprocs` is set as the least-squares method of a linear polynomial formula of the fifth order.

4.2 Examples of API Descriptions

In this section, we show examples of an API for ABCLibScript. In the following example, the software developer has a routine named EigenSolver for eigenvalue computation. The software developer can develop the new routine `foo`, which is an auto-tuning facility added to the EigenSolver routine, by using the API.

```
subroutine foo(...)
include (ABCLibScript.h)
  ...
C === Initialization and registration of AT regions.
!ABCLib$ call ABCLib_ATset (ABCLib_ALL,ABCLib_AllRoutines)
```

```
!ABCLib$ call ABCLib_ATset(ABCLib_INSTALL, ABCLib_InstallRoutines)
!ABCLib$ call ABCLib_ATset(ABCLib_STATIC, ABCLib_StaticRoutines)
!ABCLib$ call ABCLib_ATset(ABCLib_DYNAMIC, ABCLib_DynamicRoutines)
C === Perform Install-time Optimization.
C    (The following is written by the software developer.)
C     !Only one time is admitted in the while process.
!ABCLib$ call ABCLib_ATexec(ABCLib_INSTALL, ABCLib_InstallRoutines)
C        ===Only Install-time Optimization is done.
!ABCLib$ call EigenSolver(...)
  ...
C === The execution of Run-time Optimization is permitted.
C  (The following is written by the software developer.)
C   !At this time, the optimization is not executed.
C   When the target routine is called, the AT regions are optimized.
!ABCLib$ call ABCLib_ATexec (ABCLib_DYNAMIC, ABCLib_DynamicRutines)
C === When the following routine is called, Run-time Optimization is done.
C   (So, the Install-time, Before Execute-time (by the end-user),
C    and Run-time Optimizations are done in this routine.)
!ABCLib$ call EigenSolver(...)
  ...
return
end
```

The routine pooh, which is written by the end-user to use the Before Execute-time Optimization defined by the software developer, is described as follows using the API of ABCLibScript.

```
subroutine pooh(...)
include (ABCLibScript.h)
  ...
C === Perform Before Execute-time Optimization.
C  (The following is written by the end-user.)
C  !The optimization is done this time.
C      ===Fix BP values defined by software developer.
       N_TUNESIZE_START=1234
       N_TUNESIZE_END=1234
       call ABCLib_ATexec(ABCLib_STATIC, ABCLib_StaticRoutines)
C      ===Install-time and Before Execute-time Optimizations are done.
       call EigenSolver(...)
  ...
return
end
```

The example of the routine pooh indicates that the description of Before Execute-time Optimization is difficult for end-users. To reduce the work, a GUI (graphic user interface) is needed to add the function automatically.

5 Programming Examples Using ABCLibScript

The following section shows programming examples using ABCLibScript and written by the software developer.

5.1 Install-time Optimization

5.1.1 A Matrix-Matrix Multiplication Code

The following Programming Example 1 shows a program for matrix-matrix multiplication, which is applied in the adjustment function for unrolling depth in the install-time optimization.

[Programming Example 1]
Unrolling depth adjustment for matrix-matrix multiplication code.

```
!ABCLib$ install unroll (i) region start
!ABCLib$ name MyMatMul
!ABCLib$ varied (i) from 1 to 16
!ABCLib$ fitting least-squares 5
!ABCLib$ & sampled (1-5, 8, 16)
do i=1, n
  do j=1, n
    da1=A(i,j)
    do k=1, n
      dc=C(k,j)
      da1=da1+B(i,k)*dc
    enddo
    A(i,j)=da1
  enddo
enddo
!ABCLib$ install unroll (i) region end
```

In Programming Example 1, the default basic parameter BP is the variable n, which specifies the matrix dimension, because the software developer does not specify special variables for the basic parameter.

Since this example uses the instruction operator `unroll`, and specifies the loop variable i, the loop unrolling depth of the outer loop i is chosen as the performance parameter PP. By using the instruction co-operator `varied`, the range for the depth is defined from 1 to 16. The cost definition function for the execution time of the target AT region for optimizing is a fifth-order linear polynomial, which is defined by the co-operator `fitting` in this example.

In the auto-tuning system, the best values for the parameters of PP are estimated by fixing the parameters of BP, which can be specified by using the API. For the parameters of PP in this example, sampling points, which are equal to the depth of unrolling to fix the BPs, are defined by the co-operator `sampled`. In this example, the execution time from the first to the fifth, eighth, and sixteenth unrolling depths are measured. Then, based on the measured data, the coefficients of the fifth-order linear polynomial formula are determined to estimate the values for PP. The least-squares method is used in this estimation. (Also, the co-operator `least-squares` defines this.)

5.1.2 A Cache Blocking Code

Programming Example 2 shows a typical code for a block length-adjustment function for cache blocking implementation. A cache blocking algorithm is used in the kernel routine of this example.

[Programming Example 2] Adjustment of blocking length.

```
!ABCLib$ install variable (MB) region start
!ABCLib$ name BlkMatMal
!ABCLib$ varied (MB) from 1 to 64
do i=1, n, MB
  call MyBlkMatVec(A,B,C,n,i)
enddo
!ABCLib$ install variable (MB) region end
```

In Programming Example 2, the performance parameter of PP is defined as the variable of `MB` for block length by using the co-operator of `variable`. The range of the variable `MB` is from 1 to 64, and this is defined in the co-operator of `varied`.

5.2 Before Execute-time Optimization

Programming Example 3 is a program using an algorithm selection operation. The cost definition functions specified by the software developer are used as the criteria of the selection.

[Programming Example 3]
Algorithm selection based on the cost definition functions defined by the software developer.

```
!ABCLib$ static select region start
!ABCLib$ name TestSelect
!ABCLib$ parameter (in CacheS,in NB,in NPrc)
!ABCLib$   select sub region start
!ABCLib$   according to estimated (2.0d0*CacheS*NB)/(3.0d0*NPrc)
               AT Region 1
!ABCLib$   select sub region end
!ABCLib$   select sub region start
!ABCLib$   according to estimated (4.0d0*CacheS*dlog(NB))/(2.0d0*NPrc)
               AT Region 2
!ABCLib$   select sub region end
!ABCLib$ static select region end
```

In Programming Example 3, the selection of an algorithm is performed in the Before Execute-time Optimization. The selection information for AT regions 1 and 2 is parameterized as a performance parameter of PP.

The cost definition functions specified by the software developer are defined using the co-operator of `according estimated`. In this example, the floating point variables of `CacheS`, `NB`, and `NPrc`, which are defined in the Install-time optimization, are referenced. The cost of AT region 1 is estimated as $(2.0d0 * CacheS * NB)/(3.0d0 * NPrc)$, and the cost of AT region 2 is estimated as $(4.0d0 * CacheS * dlog(NB))/(2.0d0 * NPrc)$.

The evaluation of these costs is done in the Before Execute-time optimization, and then the AT region to be executed is selected in the Run-time optimization.

5.3 Run-time Optimization

Programming Example 4 shows an example for the run-time selection of AT regions, which references the variables *eps* and *iter*, which are defined in the AT regions at run-time. In this case, the best AT region which is selected minimizes the variable *eps* within $iter < 5$. This example is a typical case for adapting ABCLibScript to select pre-conditioners automatically for iterative methods.

[Program Example 4] Selection of pre-conditioners for iterative methods at run-time.

```
!ABCLib$ dynamic select (eps,iter) region start
!ABCLib$ name PricondSelect
!ABCLib$ parameter (in eps, in iter)
```

```
!ABCLib$ according to min (eps) .and. condition (iter<5)
!ABCLib$   select sub region start
             AT Region 1 (Pre-conditioner 1)
               ...
             eps = ...
!ABCLib$   select sub region end
!ABCLib$   select sub region start
             AT Region 2 (Pre-conditioner 2)
               ...
             eps = ...
!ABCLib$   select sub region end
!ABCLib$ dynamic select (eps,iter) region end
```

In Programming Example 4, the best algorithm is selected based on the floating variables *eps* and *iter* at run-time. As a performance parameter of PP, selection information for AT regions 1 and 2 is parameterized.

The cost definition function from the software developer is defined by referring to the values of the variables *eps* and *iter*, which are defined before the target AT regions run. To perform this selection, the values of *eps* are stored when the value of *iter* is less than 5 in each AT region. Then, if the value of *iter* equals 5, the minimal value is searched to select the best AT region.

6 A Test of the Pre-processor

Generally speaking, evaluating the usability of computer language is very difficult. Hence, evaluating the usability for all functions of ABCLibScript is an unrealistic task. In this section, we will evaluate limited functions of ABCLibScript.

6.1 Test for Auto-Generated Codes

We check the advantage of auto-generation for an auto-tuning facility by using ABCLibScript. This shows the advantage of automation for auto-tuning compared to programming by hand.

6.1.1 Computer Environment for the Test

We use the following computer environment in this test.

- CPU: Pentium4 (2.4 GHz)
- Memory: 256 MByte
- OS: Linux RedHat 8

- Compiler: PGI Compiler Version 4.0-2
- Compiler Option: -O0
- MPI: MPICH 1.2.1

The target process is shown as follows.

- Auto-tuning Timing: Install-time
- Unrolling Depth Specification: from 1 to 64 by using the instruction operator `unroll` in ABCLibScript.
- Target AT Region Code: the outer loop (i-loop), the second loop (j-loop), and the inner loop (k-loop) for a matrix-matrix multiplication code.
- Dimension of the matrix: 500

6.1.2 The Test Code

The following code is used in this test.

[Test Code 1] A matrix-matrix multiplication code:

```
      program  main
      include  'ABCLibScript.h'
      integer  iauto
      integer  N
      parameter (N=500)
      real*8   A(N,N), B(N,N), C(N,N)
c     === MPI Init.
c     ==================================
      call MPI_INIT( ierr )
      call MPI_COMM_RANK(MPI_COMM_WORLD,
     &          myid, ierr )
      call MPI_COMM_SIZE(MPI_COMM_WORLD,
     &           nprocs, ierr )
c     ==================================
!ABCLib$ call ABCLib_ATset(ABCLib_ALL,
!ABCLib$&   ABCLib_AllRoutines)
!ABCLib$ call ABCLib_ATset(ABCLib_INSTALL,
!ABCLib$&   ABCLib_InstallRoutines)
```

```
!ABCLib$ call ABCLib_ATset(ABCLib_STATIC,
!ABCLib$&   ABCLib_StaticRoutines)
!ABCLib$ call ABCLib_ATset(ABCLib_DYNAMIC,
!ABCLib$&   ABCLib_DynamicRoutines)
      iauto = 1
      in = 350
      if (iauto .eq. 1) then
!ABCLib$ ABCLib_NUMPROCS = 4
!ABCLib$ ABCLib_STARTTUNESIZE = 100
!ABCLib$ ABCLib_ENDTUNESIZE = 500
!ABCLib$ ABCLib_SAMPDIST = 100
!ABCLib$ call ABCLib_ATexec(ABCLib_INSTALL,
!ABCLib$&   ABCLib_InstallRoutines)
      else
!ABCLib$ ABCLib_DEBUG = 1
        call MatMul(A, B, C, in)
      endif
c     ===== MPI finazize
c     ===================================
      call MPI_FINALIZE(ierr)
c     ===================================
      stop
      end

      subroutine MatMul(A, B, C, N)
      integer N
      real*8  A(N,N), B(N,N), C(N,N)
      include  'ABCLibScript.h'
      real*8  da1, da2
      real*8  dc
      do i=1, N
        do j=1, N
          A(i,j) = 0.0d0
        enddo
      enddo
      do i=1, N
        do j=1, N
```

```
          B(j,i) = dble(i*j)
          C(j,i) = 1.0d0/dble(i*j)
        enddo
      enddo
!ABCLib$ install unroll (i) region start
!ABCLib$ name MyMatMul
!ABCLib$ varied (i) from 1 to 64
!ABCLib$ debug (pp)
      do i=1, N
        do j=1, N
          da1 = A(i,j)
          do k=1, N
           dc = C(k,j)
           da1 = da1 + B(i,k) * dc
          enddo
          A(i,j) = da1
        enddo
      enddo
!ABCLib$ install unroll (i) region end
      return
      end
```

6.1.3 Result

Test Code 1 was processed, then a code with the auto-tuning facility of FIBER was automatically generated using the pre-processor of `ABCLibCodeGen`. Fortran90 and MPI-1 were used in the automatically generated code.

We also checked the pre-processing by taking into account the target loops — the i-loop, j-loop and k-loop were unrolled in this test.

Figure 5 shows the tuning results in the FIBER install-time auto-tuning for the unrolling. From the results for the install-time auto-tuning of Figure 5, we found the following facts:

- For i-loop unrolling: The best depth was 50, and speedup was 3.1 times.
- For j-loop unrolling: The best depth was 25, and speedup was 2.1 times.
- For k-loop unrolling: The best depth was 50, and speedup was 1.1 times.

The speedups were calculated between the execution time with the depth 1 unrolling and the best depth after that.

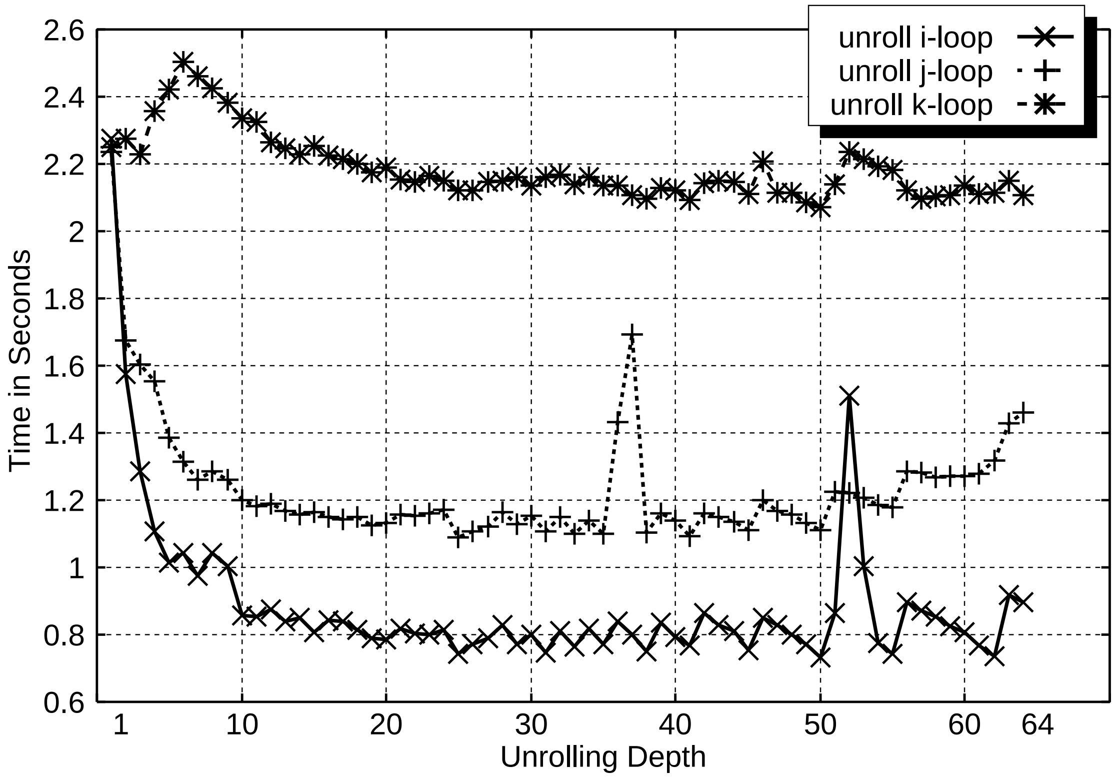

Figure 5: Auto-tuning Results for Test Code 1 (matrix-matrix multiplication.)

We found other interesting results. There were unstable phenomena in the depths of 36 and 37 for the j-loop, the depths of 51–53 for the i-loop. We think that this is caused by compiler optimization or unstable memory access, but a detailed analysis is needed. However, the important point of this test is that it is hard to find such phenomena with coding by hand, since the implementation of unrolling from 1 to 64 depths gives the software developer an enormous load. The work is a quite quick for the pre-processor.

Thus, we can check the execution and show the benefit for the directive of ABCLibScript for the auto-tuning facility of FIBER.

6.2 Test for Non-expert Users

Next, we evaluated the effect of using ABCLibScript with non-expert users.

6.2.1 Overview of the Experiment

The subjects have programming skills in C language, but have no experience using Fortran language. This experiment was done in the following two phases.

- Phase 1: Write matrix-matrix multiplication code using Fortran. Then, tune it to maximize its performance.
- Phase 2: Add ABCLibScript directives to the programs.

The span of Phase 1 was 1 week, and the span of Phase 2 was 1 day. The experiment was done from August 13th to August 20th of 2004 for Phase 1, and from August 23rd to August 24th for Phase 2.

An Intel Pentium4 (2.0G Hz) with 1 GB memory was the target architecture. The compiler was a PGI Compiler Version 4.0-2. The compiler option was set to "-O0" to evaluate the effect of code optimization by hand coding.

For the ABCLibScript directives, the following limitations were given.

- Instruction Operator: Should use `Install` and `unroll`
- Instruction Co-operator: Should use `variable`
- Other instruction operators and co-operators are not allowed to be used.
- Sampling Points of BP for Matrix Size: The points from 128 to 640 with stride 128 are set by the install-time optimization.

6.2.2 Result

[Subject A]

Figure 6 shows the performance between hand-tuning code by Subject A and auto-generated code by ABCLibScript instructed by Subject A.

Subject A wrote a simple 3-nested loop. Subject A then instructed the unrolling function by ABCLibScript to the three loop variables. The define area for the unrolling depth was from 1 to 8; hence, the generated code included 8*8*8=512 kinds of kernels. In this case, the 512 kernels were tested in the Install-time optimization.

Figure 6 shows that much speedup is obtained with ABCLibScript. Especially, the performance with ABCLibScript is stable and reaches about 110 MFLOPS. We obtained a maximum 4.3 times speedup with ABCLibScript in this case.

[Subject B]

Figure 7 shows the performance between hand-tuning code by Subject B and auto-generated code by ABCLibScript instructed by Subject B.

Subject B wrote a blocked program with the unrolled depth 2 for the innermost loop. The loop was 6-nested. Subject B instructed the unrolling function by ABCLibScript to use the variables of the 4th and 5th loops. The define area for the unrolling depth was from 1 to 16;

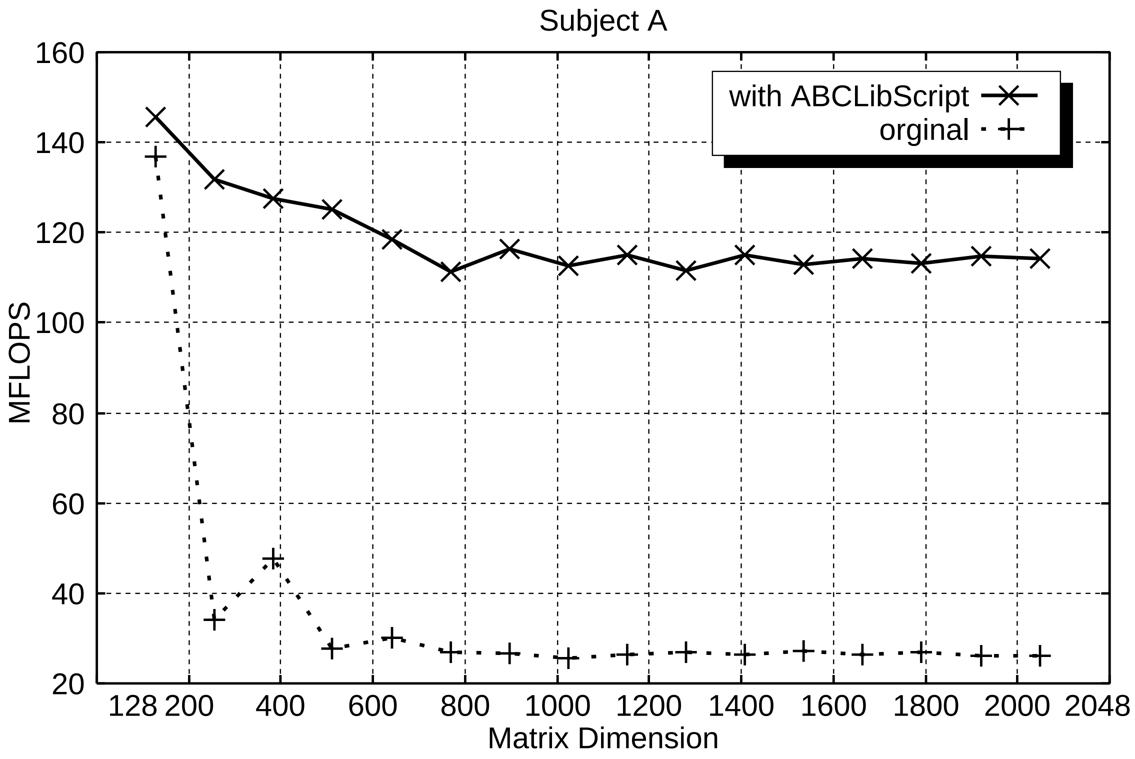

Figure 6: Performance result for Subject A (matrix-matrix multiplication.)

hence, the generated code included 16*16=256 kinds of kernels. In this case, 256 kernels were tested in the Install-time optimization.

Figure 7 shows that much speedup is obtained by using ABCLibScript. For the performance, it reaches 200 MFLOPS.

As a result of this experiment, we can say that ABCLibScript is a useful tool from the viewpoint of performance for non-expert users.

7 Related Work

We can classify conventional auto-tuning software into the following three categories.

Complete Run-time Optimization Software: In this category, the software performs the parameter adjustments at run-time. For example, to tune computer system parameters such as I/O buffer size, Active Harmony [17] and Autopilot [14] can be used. On the other hand, SANS [5] provides a framework based on run-time optimization by a network agent.

Complete Install-time Optimization Software: In this category, the software performs the parameter adjustments at the install-time. For example, PHiPAC [2], ATLAS and the paradigm of AEOS (Automated Empirical Optimization of Software) [1, 18], and FFTW [6] can auto-

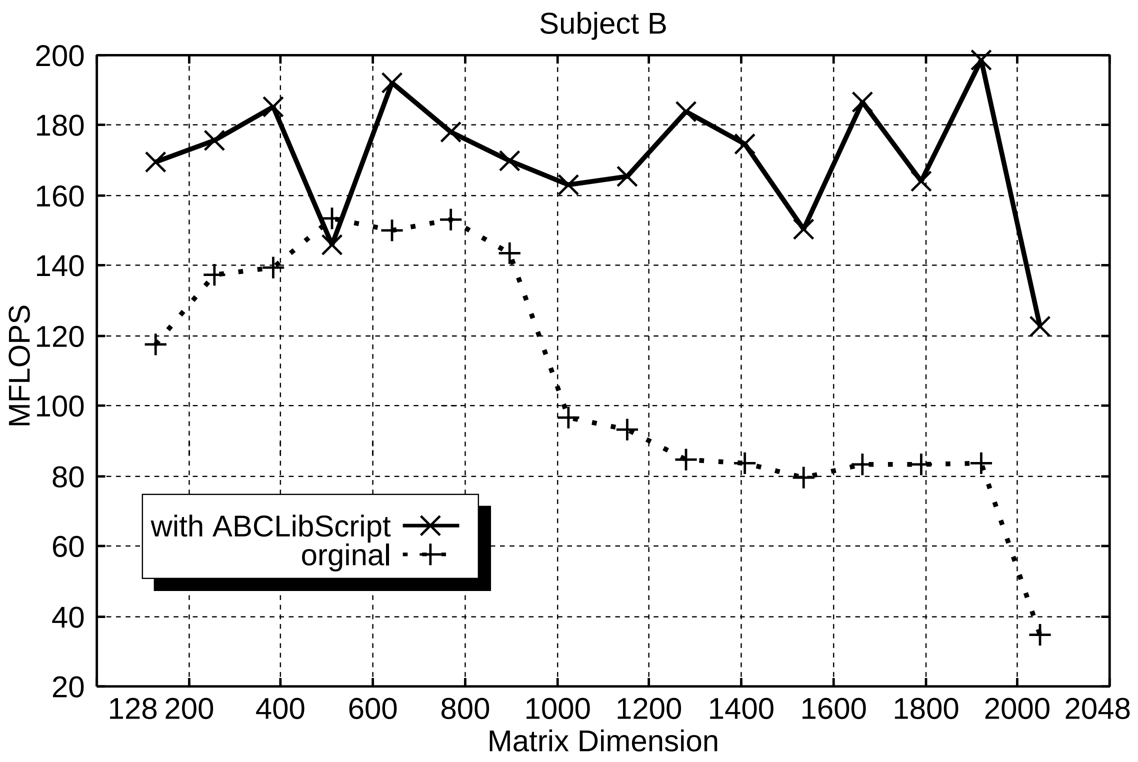

Figure 7: Performance result for Subject B (matrix-matrix multiplication.)

matically tune the performance parameters for computation kernels of their routines when they are installed. Later, in the SOLAR framework [4], the implementation of hierarchy routines for computation kernels is considered an install-time optimization.

For formalization of an auto-tuning facility in this category, Naono and Yamamoto formulated the install-time optimization in the SIMPL [13] auto-tuning software framework, which is a paradigm for parallel numerical libraries. To reduce the search time, a theory for optimal parameters in an eigensolver was studied by Imamura and Naono [7]. Their theory can be implemented as a function of ABCLibScript in a new optimization method.

Hybrid Install-time and Run-time Optimization Software: In ILIB [11, 12], the facility of install-time and run-time optimizations is implemented.

The concepts of the execute-time optimization layer with the user's knowledge, in order to improve parameter accuracy and to generalize auto-tuning facilities, are not clear and rarely discussed in the conventional auto-tuning software mentioned above. We therefore believe that BEOL in FIBER is a very new concept.

On the other hand, Brewer [3] proposed a new auto-tuning framework for a library by using the structure of the source code and measured the execution time of the codes to select the best algorithm. His frameworks were based on an automated modeling function. His method can be implemented as a new automated function for the cost definition function in FIBER; hence, the implementation will be future work.

Finally, we emphasize that there are no approaches for an easy auto-tuning facility description and no developments for a computer language pre-processor in these methods. ABCLibScript is the only language that can describe the auto-tuning facilities.

8 Conclusion Remarks

In this paper, we describe a design policy and implementation for ABCLibScript, which is a directive of the auto-tuning facility based on the FIBER framework. ABCLibScript focuses on the application of numerical software and support for the software developer, who wants to create auto-tuning software, by supplying limited functions for numerical computation.

The FIBER framework needs knowledge from two users, who are defined in the FIBER framework—that is, the software developer and the end-user [10]. From the software developer, the following knowledge is needed to describe the auto-tuning facility with ABCLibScript: (1) Extraction for effective parameters, (2) Specification of target regions, and (3) Heuristic Values, such as maximal unrolling depth or block length. From the end-user, the following are needed for auto-tuning in the Before Execute-time optimization to select the best algorithm: (1) The problem size to execute, and (2) The required accuracy.

We have developed a prototype pre-processor of ABCLibScript, which is a version with limited functions, such as the unrolling function. The prototype will be opened on the World Wide Web page of `http://www.abc-lib.org/`.

The following things should be considered as future work.

- Nested Instruction Operations: The execution model or specification when the instruction operations are nested should be considered. Generally speaking, the search space for parameters has greatly increased in this situation. The reduction or effective methods for searching, hence, should be considered in the specification of searching.

- Extension of Specification for Automatical Setting of Cost Definition Function: In the current specification of ABCLibScript, the software developer should describe the cost definition functions and assure the accuracy of the specified cost definition function. If the software developer does not know the attributes of their software, the auto-tuning facility may not work well. To remedy the situation, the automatic addition of the cost definition function is needed. Consideration of the function and the extension of the current specification for ABCLibScript will be important future work.

- Addition of Instruction Operation for Performance Stabilization: Extension of the specification for the Performance Stabilization Facility, which is proposed by Imamura and

Naono [8], should be considered as important future work.

- Extension of Specification for Grid and PDA environments: The current specification of ABCLibScript only focuses on supercomputer and heterogeneous PC cluster environments. In the FIBER project, Grid and PDA environments are targets for adapting the auto-tuning facility. Extending the specification and taking into account the nature of such environments is future work.

Acknowledgments

This study is supported by Japan Science and Technology Agency, "Information Infrastructure and Applications," PRESTO.

References

[1] Atlas project; http://www.netlib.org/atlas/index.html.

[2] Jeff Bilmes, Krste Asanović, Chee-Whye Chin, and Jim Demmel. Optimizing matrix multiply using phipac: a portable, high-performance, ansi c coding methodology. *Proceedings of International Conference on Supercomputing 97*, pages 340–347, 1997.

[3] Eric Allen Brewer. Portable high-performance supercomputing: High-level platform-dependent optimization. Technical report, Ph.D Thesis, Massachusetts Institute of Technology, 1994.

[4] Javier Cuenca, Domingo Gimenez, and Jose Gonzalez. Architecture of an automatically tuned linear algebra library. *Parallel Computing*, 30:187–210, 2004.

[5] Jack Dongarra and Vector Eijkhout. Self-adapting numerical software for next generation applications. *The International Journal of High Performance Computing and Applications*, 17(2):125–131, 2003.

[6] Matteo Frigo. A fast fourier transform compiler. In *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 169–180, Atlanta, Georgia, May 1999.

[7] Toshiyuki Imamura and Ken Naono. An evaluation towards an automatic tuning eigensolver with performance stability. In *Proceedings of Symposium on Advanced Computing Systems and Infrastructures (SACSIS)2003*, pages 145–152, 2003.

[8] Toshiyuki Imamura and Ken Naono. Development of a numerical library with a performance stabilizing mechanism for cache conflicts. In *Proceedings of High Performance Computing Symposium (HPCS) 2004*, pages 173–180, 2004.

[9] Takahiro Katagiri, Kenji Kise, Hiroki Honda, and Toshitsugu Yuba. Fiber : A software framework to support automatically addition of generalized auto-tuning facilities. *IPSJ SIG Technical Report*, 2003-HPC-94:1–6, 2003.

[10] Takahiro Katagiri, Kenji Kise, Hiroki Honda, and Toshitsugu Yuba. Effect of auto-tuning with user's knowledge for numerical software. In *Proceedings of ACM Computing Frontiers 04*, pages 12–25, Island of Ischia, Italy, April 2004.

[11] Takahiro Katagiri, Hisayasu Kuroda, Kiyoshi Ohsawa, Makoto Kudoh, and Yasumasa Kanada. Impact of auto-tuning facilities for parallel numerical library. *IPSJ Transaction on High Performance Computing Systems*, 42(SIG 12 (HPS 4)):60–76, 2001.

[12] Hisayasu Kuroda, Takahiro Katagiri, and Yasumasa Kanada. Knowledge discovery in auto-tuning parallel numerical library. *Progress in Discovery Science, Final Report of the Japanese Discovery Science Project, Lecture Notes in Computer Science*, 2281:628–639, 2002.

[13] Ken Naono and Yuusaku Yamamoto. A framework for development of the library for massively parallel processors with auto-tuning function and with the single memory interface. *IPSJ SIG Notes*, (2001-HPC-87):25–30, 2001.

[14] Randy L. Ribler, Huseyin Simitci, and Daniel A. Reed. The autopilot performance-directed adaptive control system. *Future Generation Computer Systems, special issue (Performance Data Mining)*, 18(1):175–187, 2001.

[15] Katagiri Takahiro, Kinji Kise, Hiroaki Honda, and Toshitsugu Yuba. Fiber: A general framework for auto-tuning software. *Proceedings of The Fifth International Symposium on High Performance Computing*, Springer Lecture Notes in Computer Science(2858):146–159, 2003.

[16] Katagiri Takahiro, Kinji Kise, Hiroaki Honda, and Toshitsugu Yuba. Fiber: A framework of installation, before execution-invocation, and run-time optimization layers for auto-tuning software. *IS Technical Report, Graduate School of Information Systems, The University of Electro-Communications*, UEC-IS-2003-3, May 2003.

[17] Cristian Tapus, I-Hsin Chung, and Jeffery K. Hollingsworth. Active harmony : Towards automated performance tuning. In *Proceedings of High Performance Networking and Computing (SC2002)*, Baltimore, USA, November 2003.

[18] R.Clint Whaley, Antoone Petitet, and Jack J. Dongarra. Automated empirical optimizations of software and the atlas project. *Parallel Computing*, 27:3–35, 2001.