Combining Code Refactoring and Auto-Tuning to Improve Performance Portability of High-Performance Computing Applications

Chunyan Wang*†, Shoichi Hirasawa*†, Hiroyuki Takizawa*†, and Hiroaki Kobayashi‡
* Graduate School of Information Sciences, Tohoku University
Sendai, Japan
Email: {wchunyan, hirasawa}@sc.isc.tohoku.ac.jp, takizawa@cc.tohoku.ac.jp
†CREST, Japan Science and Technology Agency
‡Cyberscience Center, Tohoku University
Email: koba@cc.tohoku.ac.jp

***Abstract*—An existing High-Performance Computing (HPC) application is usually optimized for a particular platform to achieve high performance. Hence, such an application is often unable to run efficiently on other platforms, i. e., its performance is not portable. The purpose of this work is to establish a systematic way to improve performance portability of HPC applications, to which various kinds of platform-specific optimizations have already been applied. To this end, we combine code refactoring and auto-tuning technologies, and develop a programming tool for HPC refactoring. Auto-tuning is a promising technology to enable an HPC application to adapt to different platforms. In general, however, an auto-tuning tool is not applicable to an HPC application if the application has already been optimized for a particular platform. In this work, a code refactoring tool that interactively asks a user for necessary information to undo some platform-specific optimizations in an existing application is developed based on Eclipse, and hence auto-tuning techniques can be applied to the application. The evaluation results demonstrate that combining code refactoring and auto-tuning is a promising way to replace platform-specific optimizations with auto-tuning annotations, and thereby to improve the performance portability of an HPC application.**

Keywords–auto-tuning; code refactoring; high-performance computing; performance portability.

I. INTRODUCTION

Legacy applications are successful and therefore mature, and likely have been in existence for a long period of time. Thus, legacy applications are often required to migrate to new platforms, to use new algorithms, and to be components in larger systems. Typical platforms are determined by a hardware architecture, an operating system, and runtime libraries. Moore's law [1] has predicted the dramatic improvement of computing hardware. As legacy applications are ported to meet ever-changing requirements, even well-designed applications are subject to structural erosion. The quality of any code base with a long lifespan tends to degrade over time [2]. Maintenance of a legacy application can become an error-prone, time- and resource-consuming work.

In the specific field of HPC, the main concern is to fully utilize the potential of a specific target platform to achieve high performance [3]. Today, an existing HPC application is usually optimized for a particular platform. While HPC systems (hardware and software) reach unprecedented levels of complexity, such an application is often unable to run efficiently on other platforms because different platforms require different optimizations, and hence its performance is not portable. In order to maximize performance, the developer must carefully consider optimizations relevant to each target platform. As a result, tuning for an individual platform is a highly-specialized and time-intensive process. The increasing complexity of HPC systems, their long lifespans, and the plethora of desirable source code optimizations make HPC applications hard to maintain.

Automatic performance tuning, also known as auto-tuning, provides performance portability, as the auto-tuning process can easily be re-run on new platforms which require different sets of optimizations [4]. Auto-tuning is increasingly being used in the domain of HPC to optimize programs. However, auto-tuning is usually designed for programs not optimized for any specific platform. It is difficult to auto-tune an HPC application, to which platform-specific optimizations are already applied.

Refactoring is a promising solution to gradual software decay [2][5]. An application code can be refactored so that it is easier to apply auto-tuning technologies. Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior [5]. Refactoring tools that can automate the refactoring process have shown their advantages in improving readability, maintainability and extensibility of object-oriented programs. Behavior-preserving code transformations, which are simply called *refactorings* in the research field of code refactoring, can help to improve the code structure, thereby potentially setting the stage for improvements. Despite the potential, refactorings that are specific to HPC applications are rarely provided.

However, some refactorings for HPC applications cannot be fully automated because the information required for those refactorings has been lost when the platform-specific optimizations were applied to the application code. Users have to specify where and how the application code was optimized. Usually, a significant part of the knowledge required to perform the refactoring remains implicitly in the user's head. It is promising to use user knowledge to compensate the lost information that is necessary for the refactoring. Semi-automated refactoring can be a promising approach to refactoring application codes to be tunable with user knowledge.

Photran [6] provides some refactorings, called performance refactorings, to facilitate some loop optimizations (*interchange loop*, *fuse loop*, *reverse loop*, *tile loop* and *unroll loop*) specific to Fortran, which is the backbone of the HPC community [7]. On one hand, these refactorings help to improve the perfor-

 ISBN: 978-1-61208-394-0

mance of an application. On the other hand, loop refactorings such as loop tiling and loop unrolling make the code difficult to understand, and consequently make it hard for further optimization. Due to the differences in system configurations, an appropriate loop optimization on an HPC application usually changes depending on its target platform [8]. When the application code is ported to a newly available platform, the performance is usually not portable to the new platform. To make matters worse, it is difficult to optimize the refactored code for a new platform. Therefore, those performance refactorings always lead to low performance portability.

The purpose of this work is to establish a systematic way to improve performance portability of HPC applications, to which various kinds of platform-specific optimizations have already been applied. To achieve this goal, we combine code refactoring and auto-tuning technologies. A code refactoring tool is designed to support the process of undoing platform-specific optimizations of an existing HPC application. Note that some information such as the original loop length may be lost by platform-specific loop optimizations such as Photran's performance refactorings. Hence, we develop a refactoring tool that is assumed to be a part of an integrated development environment (IDE) so that the tool can interactively ask the user to specify the missing information for "reverse transformations" of performance refactorings. In this work, the reverse transformations are called *HPC refactorings*. As a result of HPC refactorings, platform-specific optimizations are replaced with annotations for auto-tuning to make the application adaptable to different platforms. Moreover, the resulting application code becomes easy to read and maintain. The evaluation results indicate that the HPC refactoring tool is helpful to support replacement of platform-specific optimizations with auto-tuning annotations, and thereby to improve the performance portability of an HPC application.

The remainder of this paper is organized as follows. Section II introduces the related work. Section III describes the proposed method and illustrates it with two examples. Section IV shows the evaluation of the proposed method. Finally, Section V gives the conclusion of this work, and states future work.

II. Related Work

This section reviews the related researches, and classifies them into four categories: (1) automation of refactorings; (2) code refactorings for HPC applications; (3) platform-specific optimizations that degrade performance portability; and (4) refactoring and performance tuning.

A. Fully-automated versus Semi-automated Refactoring Tools

Mens and Tourwé [9] identified three activities associated with the process of refactoring:

1) Identification of where an application code should be refactored.
2) Determination of which refactoring(s) should be applied.
3) Application of the selected refactorings.

The degree of the automation of a refactoring tool depends on which of the refactoring activities are supported by the tool.

Contemporary IDEs such as Eclipse [10] often support a semi-automated approach to refactoring. Tokuda and Batory's research [11] indicated that a semi-automated approach can drastically increase the productivity in comparison with manual refactoring.

Some researchers demonstrated the feasibility of fully-automated refactoring [12][13]. Guru is a fully automated tool for refactoring inheritance hierarchies and refactoring methods in SELF programs [13]. Optimization techniques that are performed by compilers can also be considered as fully automated refactoring techniques. Although fully automated tools provide the ability to quickly and safely improve the structure of the code without altering its functionality, the lack of user input leads to the introduction of meaningless identifiers and could make the current application become more difficult to understand than before.

Semi-automated refactoring tools involve human interaction to address the problems caused by fully-automated refactoring. The semi-automated refactoring tool may become time-consuming when the application is in large scale. Despite this problem, semi-automated refactoring remains the most useful approach in practice, since a significant part of the knowledge required to perform the refactoring cannot be extracted from the software, but remains implicit in the developer's head [9].

B. Refactorings for HPC Applications

What HPC programmers concern most is to maximize the performance on their target platforms. While refactorings are typically used to improve the readability and maintainability of a code, refactorings specific to Fortran and HPC to improve performance are rarely provided. HPC programs are more difficult to restructure because data flow and control flow are tightly interwoven [9]. Because of this, restructurings are typically limited to the level of a function or a block of code [14].

Nevertheless, many refactorings that improve performance still have to be done manually. The parser and general language infrastructure of a refactoring tool and performance preserving requirements make it still a great challenge to develop a new refactoring tool for the specific domain of HPC. Great efforts have been made to develop tools to restructure the Fortran and HPC codes [3][15].

Bodin et al. built an object-oriented toolkit and class library for building Fortran and C++ restructuring tools [16]. It requires complete understanding of the internal parser structures for users to add language extensions to Fortran or C. CamFort [17] is a tool that provides automatic refactoring for improving the code quality of existing models. SPAG [18] is a restructuring tool, which unscrambles spaghetti Fortran66 code, and convert it to structured Fortran 77. New projects such as Eclipse Parallel Tools Platform (PTP) have recently made strides to address the needs of HPC developers [15]. It provides some refactorings such as Rename and Extract Method. As a component of the Eclipse PTP, Photran provides an IDE and refactoring tool for Fortran. However, since Photran was originally designed to improve maintainability or performance [19], it does not go far enough towards solving the performance portability problem.

C. Platform-Specific Optimizations

Some researchers have indicated that platform-specific optimizations degrade the performance portability of application codes. Ratzinger et al. [20] exploited historical data extracted from repositories, and pointed out that certain design fragments in software architectures can have a negative impact on system maintainability. Then, they proposed an approach to detecting

 ISBN: 978-1-61208-394-0

such design problems to improve the evolvability of an application. Bailey et al. [21] pointed out that, an application may not be able to run efficiently with available computer resource unless the application has been optimized for the particular system. Our previous research [22] has also reported that platform-specific optimizations have a strong negative impact on performance portability of an existing HPC application.

D. Refactoring and Performance Tuning

Fowler [5] stated that refactoring certainly will make software go more slowly, but it also makes the software more amenable to performance tuning. Du et al. [23] claimed that if the architectural details are known, auto-tuning is an effective way to generate tuned kernels that deliver acceptable levels of performance. Moore's research [24] applied *Extract Procedure* refactoring to the performance critical regions to facilitate performance tuning. The research also shows the possibility to enhance refactoring tools with auto-tuning techniques. In [25], it is verified that OpenACC directives for accelerators provide a mechanism to stay in a current high-level language. OpenACC directives are expected to enable programmers to easily develop portable applications that maximize the performance with the hybrid CPU/GPU architecture, and accelerate existing applications quickly by adding a few lines of code. However, platform-specific optimizations are still required even if OpenACC directives are adopted for accelerator computing [26].

III. The Proposed HPC Refactoring Tool

In this work, we develop an HPC refactoring tool to improve performance portability of HPC applications. Section III describes the methodology of the proposed HPC refactoring tool. This method provides an undo mechanism with code refactoring to undo platform-specific optimizations to make an application code tunable, and a redo mechanism of the optimizations with an auto-tuning technique. Undoing of optimizations is necessary because auto-tuning tools usually assume un-optimized codes as their inputs. By using auto-tuning to make an existing HPC application adaptive to other platforms, the performance portability of the application can be improved in a systematic way.

A. Overview of the Proposed Method

Given an application that is optimized for a specific platform for high performance, the performance portability of this application is the ability to retain the performance when the application is ported to other platforms.

Figure 1 illustrates the proposed method to improve performance portability. *Original Program* represents the source program that has already been optimized for a specific platform. In this work, we develop a code refactoring tool as a plug-in of Eclipse IDE to automate the process of undoing platform-specific optimizations such as loop unrolling with a certain unroll factor. With our refactoring tool, *Original Program* is refactored to *Refactored Program*, in which platform-specific optimizations are undone. Hence, an auto-tuning technique can be applied to redo the optimizations to achieve high performance on various platforms.

While it is difficult to auto-tune *Original Program*, *Refactored Program* can use an auto-tuning technique because such a technique is usually designed for programs not optimized for a specific platform. As a result, *Autotuned Program* that is *Refactored Program* with an auto-tuning technique can

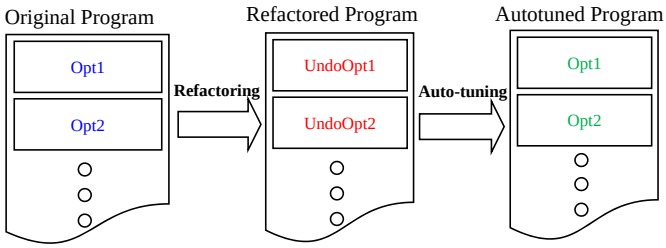


Figure 1. Overview of the proposed method for improving performance portability.

achieve high performance portability across multiple platforms while maintaining the performance on the original platform, for which *Original Program* was optimized. In this way, we systematically improve the performance portability of an existing HPC application.

The code refactoring tool can undo a platform-specific optimization only if the transformation rule of the optimization is already known. Performance refactorings, such as loop tiling and loop unrolling, provided by Photran degrade the performance portability of an application code. Since the transformation rule of these refactorings are already known, we will use these refactorings to explain the undo mechanism and show how the proposed method can contribute to the performance portability.

B. Undo Mechanism of the Proposed Method

The undo mechanism provides a semi-automated refactoring tool to allow users to specify the necessary information for refactoring. Users select the code region and determine which refactoring should be applied. The refactoring tool then checks the preconditions for behavior preserving. Once the preconditions are guaranteed, users are asked to input necessary information for applying corresponding refactoring.

1) Undo Loop Unrolling: Loop unrolling is a loop transformation technique that attempts to optimize a program's performance by reducing instructions that control the loop [27].

The general form of the do-loop is as follows.

```
do var = expr1, expr2, expr3
   statements
end do
```

Here, *var* is the loop index, *expr1* specifies the initial value of *var*, *expr1* is the upper bound, and *expr3* is the increment (step). The variable defined in the do-statements is incremented by 1 by default.

Loop unrolling replicates the code inside a loop body multiple times. The *step* that determines the number of replications is called an *unroll factor*.

The *unroll loop* refactoring provided by Photran applies loop unrolling to the outermost loop of the selected nested do-loop. Figure 2 shows a typical do-loop unrolled by *unroll loop* refactoring when the selected do-loop is the innermost loop and the unroll factor is set to be "*B*."

To perform an undoing operation, users have to specify the do-loop nest that loop unrolling was applied to and the unroll factor prior to the unrolled loop nest. Then the loop header is rewritten according to the user input, and the duplicated statements are removed. The user should input the necessary information in the following format.

 ISBN: 978-1-61208-394-0

```
do j=1,N
 do k=1,N
  do i=1,N,B
   c(i,j)=c(i,j)+a(i,k)*b(k,j)
   if(i+1>N) exit
   c((i+1),j)=c((i+1),j)+a((i+1),k)*b(k,j)
   if(i+2>N) exit
   c((i+2),j)=c((i+2),j)+a((i+2),k)*b(k,j)
   ...
   if(i+B-1>N) exit
   c((i+B-1),j)=c((i+B-1),j)+a((i+B-1),k)*b(k,j)
  end do
 end do
end do
```

Figure 2. A do-loop unrolled B times by *unroll loop* refactoring in Photran.

```
!$Undo unroll(factor)
```

!$Undo unroll specifies the refactoring that should be performed, and argument *factor* indicates the unroll factor that was applied to the loop nest. The new step value will be the current step value (*B* for example in Figure 2) divided by *factor*. As for the duplicated statements removal, it is assumed that the number of statements is a multiple of the unroll factor. The number of statements remaining after undoing loop unrolling, called *NoofStat*, is *the current number of statements* divided by *factor*. The first *NoofStat* statements in the loop body will be kept, and the rest statements will be removed.

With the help of our refactoring tool, the platform-specific optimization, e. g., loop unrolling applied to the code in Figure 2, is undone. Figure 3 shows the resulting code by applying the undo mechanism to the example shown in Figure 2. In this case, the loop variable is set to increment by 1 and the duplicated loop body is removed.

```
do j=1,N
 do k=1,N
  do i=1,N
   c(i,j)=c(i,j)+a(i,k)*b(k,j)
  end do
 end do
end do
```

Figure 3. The resulting do-loop after applying undo mechanism.

2) Undo Loop Tiling: Loop tiling partitions a loop's iteration into smaller chunks or blocks, thus helps eliminate as many cache misses as possible, and maximize data reuse [28].

The *tile loop* refactoring in Photran takes a double-nested do-loop of unit-step loops, and creates a nested do-loop with four levels of depth. This refactoring requires a user to provide inputs of the tile size and the tile offset, so that the user can set those parameters according to her/his will. The tile size determines the size of the accessing block. For instance, if the tile size is 3, an array will be accessed in 3×3 blocks. The tile offset adjusts where the blocks start.

Let *loopBound* and *newBound* represent the loop bounds of the double-nested do-loop before and after loop tiling, respectively. The new bounds are computed using Equation (1).

$$newBound = floor((loopBound - tileOffset) / tileSize) * tileSize + tileOffset \quad (1)$$

Figure 4 shows a typical do-loop format before loop tiling. Given a tile size "*IS*" and tile offset "*offset*," the *tile loop* refactoring in Photran always generates a tiled loop with the following format shown in Figure 5. The outer loop goes over the "blocks" and the inner loop traverses each block in its turn. Block size *IS* should be chose to fit in the cache or a memory page (whichever is smaller).

```
!before loop tiling
do j=1,N
 do i=1,N
  a(i,j)=a(i,j)+b(i,j)*a(i,j)
 end do
end do
```

Figure 4. A typical do-loop before loop tiling.

```
!after loop tiling
do j1=j_newLb, j_newUb, IS
 do i1=i_newLb, i_newUb, IS
  do j=max(j1,1), min(N, j1+IS-1)
   do i=max(i1,1), min(N, i1+IS-1)
    a(i,j)=a(i,j)+b(i,j)*a(i,j)
   end do
  end do
 end do
end do
```

Figure 5. A do-loop tiled by *tile loop* refactoring in Photran with tile size *IS*.

To perform the undo mechanism of loop tiling to a do-loop nest whose loop header has the format as shown in Figure 5, users are required to specify the do-loop nest that loop tiling was applied to and give the loop bounds for the new do-loop nest. Users should provide the necessary information in the following format prior to the do-loop nest.

```
!$Undo tile(loopName1,start1, end1,
  loopName2, start2, end2)
```

!$Undo tile specifies the refactoring that is going to be performed, and arguments *(loopName1,start1, end1, loopName2, start2, end2)* specify the new loop index variables and corresponding loop bounds. When the loop index matches the information from users and its step value does not equal to 1, its loop header is replaced as the users specified. The loop header whose index does not match the user information is removed.

With the help of our refactoring tool, the nested do-loop "after loop tiling" can be restored to that "before loop tiling." In this way, the proposed method undoes the loop tiling that has been applied to the program. Hence, the modified program can be tunable for further optimization by using auto-tuning techniques developed for un-optimized codes.

 ISBN: 978-1-61208-394-0

C. Redo Mechanism of Proposed Method

The redo mechanism is supported by an auto-tuning technique. This paper assumes auto-tuning based on full parameter search that tests all code variants of an annotated code fragment and selects the best one with the highest performance, even though any auto-tuning tools, such as the ROSE auto-tuning mechanism [29] can be employed in the proposed method. An annotation-based code generator, such as HMPPCG [30], is assumed to generate the code variants.

Generally, an auto-tuning tool is designed for un-optimized codes. Our refactoring tool, which undoes platform-specific optimizations, can refactor the *Original Program* to be tunable, thus the *Refactored Program* can easily incorporate the auto-tuning tool to obtain the best parameters for each platform. As a result, *Autotuned Program* becomes adaptable to each platform. By combining code refactoring and auto-tuning, an existing HPC application can become adaptable to different platforms. Accordingly, its performance portability is improved in a systematic way.

IV. Performance Evaluation

This section shows the evaluation of our refactoring tool and illustrates the benefit of the proposed method. The undo mechanism is supported by a code refactoring tool, which is developed by considering the Eclipse IDE [10] in mind so that the IDE can provide an interactive user interface to ask the user about necessary information for HPC refactoring. The redo mechanism is supported by an empirical auto-tuning technique that can search for the optimal tuning parameter for each different platform.

The performance of *Original Program* on the specific CPU platform, for which the program has originally been optimized, is taken as the baseline performance on each platform. We apply the proposed undo mechanism to *Original Program* and obtain *Refactored Program* whose kernel code is tunable. An auto-tuning technique based on full parameter search is applied to *Refactored Program*, and thus we can get *Autotuned Program* that has been adapted to another platform by using the auto-tuning technique. Performance portability is discussed according to the evaluation results.

A. Experimental Setup

To validate the effectiveness of the proposed method, we measure the execution performance of a program on different platforms to evaluate the performance portability. Platforms with different cache sizes used in the following evaluation are listed in Table I.

To show the effects of optimizations more clearly, we used the "-O0" option for the GNU compiler to disable compiler's optimizations. For the programs running on the SX-9 system, we used the "-O nounroll" option to disable the automatic unrolling optimization of the FORTRAN90/SX compiler. To evaluate the effects of HPC refactoring of unrolled loops, Fortran matrix multiplication programs of a simple triple-nested loop are used for multiplying two matrices of 512×512. With the consideration of fitting the accessed array elements into last level cache, we change the matrix size to be 3500×3500 to evaluate HPC refactoring of tiled loops.

B. Results and Discussions

Auto-tuning based on full parameter search is used to find the optimal unroll factor and tile size for each target platform. The execution time of the original program whose unroll factor and tile size are optimized for Intel Core i7 930 are evaluated on the four platforms listed in Table I. As shown in Figure 6, the execution time changes with the unroll factor and tile size on each platform. The experimental results indicate that the unroll factor and tile size have a considerable impact on performance of different platforms. Thus, the optimal parameters can be different for individual platforms with different configurations. It is necessary to tune those parameters to achieve high performance on each target platform. Therefore, auto-tuning is needed to achieve high performance portability. The experiment based on empirical auto-tuning found that the optimal unroll factors for platforms 1 to 4 are 64, 64, 16 and 1, respectively. The optimal tile sizes for platforms 1 to 4 are 1201, 1280, 720 and 256, respectively. These optimal parameters are used for further experiments to evaluate the performance portability.

It is shown that, the performance of the SX-9 system is sensitive to the parameter configuration. This is because the *Original Program* is optimized for a totally different system with Intel Core i7. As a simple matrix multiplication program of a triple-nested loop is used in the evaluation, the sustained performance of each platform is thus limited by the cache size and memory bandwidth. The SX-9 system should achieve a much higher performance than the others. The results indicate that inappropriate optimizations mismatching the system architecture could critically degrade the sustained performance, even if the theoretical performance is high. These results hence show the importance of making an HPC application adaptive to other platforms to cope with system diversity.

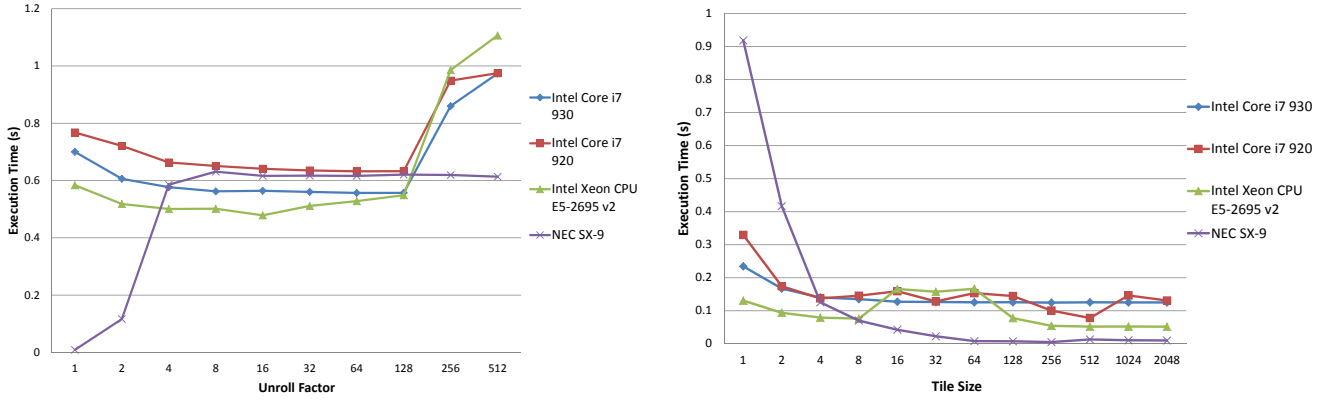
To validate the usability of the proposed method, we measure the speedup ratios of target programs on each platform to evaluate the performance portability. The evaluation results are shown in Figure 7. In this figure, the horizontal axis shows the four target platforms, and the vertical axis shows the speedup on each target platform. "*Original Program*" represents the program whose unroll factor and tile size are optimized for Intel Core i7 930. "*Refactored Program*" indicates the program after undoing the optimizations with our refactoring tool. "*Autotuned Program* " shows the program that is optimized for each platform with the optimal unroll factor and tile size obtained from the previous evaluation.

In Figure 7, *Autotuned Program* can achieve the same performance as *Original Program* on the original target platform, i .e., Intel Core i7 930. Even though *Original Program* can get fair or higher performance on platforms with high peak computational performance than Intel Core i7 930, the computational ability of each platform is not well utilized, and the performance portability of *Original Program* is low. *Refactored Program* can incorporate auto-tuning more easily than *Original Program* with the help of our refactoring tool. Thus, *Autotuned Program* can achieve comparable to or higher performance than *Original Program* on other platforms by auto-tuning optimization parameters. Accordingly, by replacing platform-specific optimizations with auto-tuning annotations, an HPC application can be performance-tunable and its performance becomes, at a certain level, portable to other platforms. These results clearly indicate the effectiveness of our method to improve the performance portability. The refactoring tools will be helpful to easily and safely do the refactoring.

 ISBN: 978-1-61208-394-0

TABLE I. EXPERIMENTAL ENVIRONMENT.

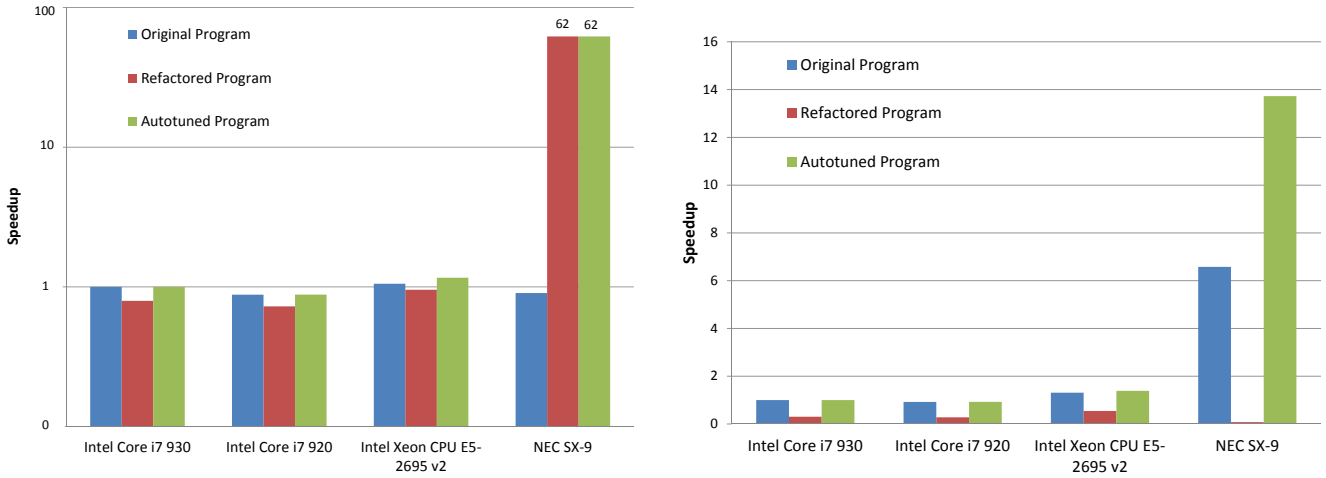
Platform	*CPU*	*Last Level Cache*	*Peak Computational Performance (Gflops)*	*Maximum Memory Bandwidth (GB/s)*
1	Intel Core i7 930	8MB	51.2	25.6
2	Intel Core i7 920	8MB	42.56	25.6
3	Intel Xeon CPU E5-2695 v2	30MB	230.4	59.7
4	NEC SX-9	256KB	102.4	256



(a) Execution time changes with different unroll factors.

(b) Execution time changes with different tile sizes.

Figure 6. Execution performance evaluation with different parameter configurations.



(a) Speedup for Loop Unrolling.

(b) Speedup for Loop Tiling.

Figure 7. Performance evaluation results by applying the proposed method. Original Program is optimized for Intel Core i7 930.

Since platform-specific optimizations are removed from *Original Program*, *Refactored Program* runs slower than *Original Program*. The refactoring of undoing loop unrolling brings performance optimization on the NEC SX-9 platform. The vectorization processing on SX-9 is considered to be the reason of performance improvement.

The refactoring of undoing loop tiling optimization degrades the performance significantly on the NEC SX-9 platform. NEC SX-9 is equipped with an on-chip memory of 256KB called ADB (Assignable Data Buffer) [31] to realize a higher memory bandwidth as well as a shorter latency on a chip for efficient vector data accesses. The performance on SX-9 is significantly affected by the ADB size. This indicates that the appropriate optimization is crucial to the performance on different platform. The platform-specific optimizations in an application should be removed to improve the performance portability.

V. CONCLUSION AND FUTURE WORK

This paper has proposed a systematic way for improving performance portability of HPC applications by combining code refactoring and auto-tuning technologies. A semi-automated code refactoring tool that uses user knowledge is developed as an Eclipse plug-in to support undo platform-specific optimizations so that the HPC applications can be refactored to

 ISBN: 978-1-61208-394-0

be tunable. Then, an auto-tuning technique is applied to *Refactored Program* to redo optimizations in different ways so that the application can adapt to multiple platforms. Therefore, the performance portability of an existing HPC application can be improved. The evaluation results demonstrate that combining code refactoring and auto-tuning is a promising way to replace platform-specific optimizations with auto-tuning annotations, and thereby to improve the performance portability of an existing HPC application.

This paper considered that only one kind of optimizations is applied to the selected code region. However, in practice, multiple optimizations can be applied to the same code region. In the future, we will explore more complicated codes in which multiple optimizations are applied.

ACKNOWLEDGMENT

This work was partially supported by JST CREST "An Evolutionary Approach to Construction of a Software Development Environment for Massively-Parallel Heterogeneous Systems" and Grant-in-Aid for Scientific Research (B) #25280041.

REFERENCES

[1] C. A. Mack, "Fifty years of Moore's law," IEEE Transactions on Semiconductor Manufacturing, vol. 24, no. 2, May 2011, pp. 202–207.

[2] J. Overbey, S. Xanthos, R. Johnson, and B. Foote, "Refactorings for Fortran and high-performance computing," in Proceedings of the Second International Workshop on Software Engineering for High Performance Computing System Applications, ser. SE-HPCS'05. New York, USA: ACM, 2005, pp. 37–39.

[3] R. Johnson, B. Foote, J. Overbey, and S. Xanthos, "Changing the face of high-performance Fortran code," White Paper, pp. 1–9, January 2006.

[4] J. Demmel, S. Williams, and K. Yelick, "Automatic performance tuning (autotuning)," in The Berkeley Par Lab: Progress in the Parallel Computing Landscape, D. Patterson, D. Gannon, and M. Wrinn, Eds. Microsoft Research, August 2013, pp. 337–339.

[5] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, Refactoring: Improving the Design of Existing Code. Addison-Wesley, 1999, iSBN-10:0-201-48567-2.

[6] "Photran 7.0 advanced features," The Elipse Foundation, 2011. [Online]. Available: https://wiki.eclipse.org/PTP/photran/documentation/photran7advanced [retrieved: 2015.02.09]

[7] F. G. Tinetti and M. Méndez, "Fortran legacy software: Source code update and possible parallelisation issues," SIGPLAN Fortran Forum, vol. 31, no. 1, March 2012, pp. 5–22.

[8] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, "Optimization of sparse matrix-vector multiplication on emerging multicore platforms," in Proceedings of the 2007 ACM/IEEE Conference on Supercomputing (SC'07), November 2007, pp. 1–12.

[9] T. Mens and T. Tourwé, "A survey of software refactoring," IEEE Transactions on Software Engineering, vol. 30, no. 2, 2004, pp. 126–139.

[10] G. Watson, J. Alameda, B. Tibbitts, and J. Overbey, "Developing scientific applications using Eclipse and the parallel tools platform," 2010. [Online]. Available: http://download.eclipse.org/tools/ptp/docs/ptp-sc10-tutorial.pdf [retrieved: 2015.02.09]

[11] L. Tokuda and D. Batory, "Evolving object-oriented designs with refactorings," in Automated Software Engineering, ser. 1, vol. 8. Kluwer Academic Publishers, 2001, pp. 89–120.

[12] T. M. Peter Ebraert, Theo D'Hondt, "Enabling dynamic software evolution through automatic refactoring," in Proceedings of the 1st Int'l Workshop on Software Evolution Transformations, November 2004, pp. 3–7.

[13] I. Moore, "Guru - a tool for automatic restructuring of self inheritance hierarchies," in Technology of Object-Oriented Language Systems (TOOLS), vol. 17, 1995, pp. 267–275.

[14] J. L. Overbey, S. Negara, and R. E. Johnson, "Refactoring and the evolution of Fortran," in Proceedings of the 2009 ICSE Workshop on Software Engineering for Computational Science and Engineering, ser. SECSE'09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 28–34.

[15] F. B. Kjolstad, D. Dig, and M. Snir, "Bringing the HPC programmer's ide into the 21st century through refactoring," in SPLASH 2010 Workshop on Concurrency for the Application Programmer (CAP'10). Association for Computing Machinery (ACM), October 2010, pp. 1–4.

[16] F. Bodin et al., "Sage++: An object-oriented toolkit and class library for building Fortran and C++ restructuring tools," in The second annual object-oriented numerics conference (OON-SKI, 1994, pp. 122–136.

[17] D. Orchard and A. Rice, "Upgrading Fortran source code using automatic refactoring," in Proceedings of the 2013 ACM Workshop on Workshop on Refactoring Tools, ser. WRT'13. New York, USA: ACM, 2013, pp. 29–32.

[18] "SPAG - Fortran code restructuring," Polyhedron Software Products, 2014. [Online]. Available: http://www.polyhedron.com/products/fortran-tools/plusfort-with-spag/spag-fortran-code-restructuring.html [retrieved: 2015.02.09]

[19] M. Méndez, J. Overbey, A. Garrido, F. G. Tinetti, and R. Johnson, "A catalog and classification of Fortran refactorings," in In 11th Argentine Symposium on Software Engineering (ASSE 2010), 2010, pp. 500–505.

[20] J. Ratzinger, M. Fischer, and H. Gall, "Improving evolvability through refactoring," SIGSOFT Softw. Eng. Notes, vol. 30, no. 4, May 2005, pp. 1–5.

[21] D. Bailey, R. Lucas, and S. Williams, Performance Tuning of Scientific Applications, ser. Chapman & Hall/CRC Computational Science. CRC Press, 2010.

[22] C. Wang, S. Hirasawa, H. Takizawa, and H. Kobayashi, "Code refactoring for high performance computing applications," in Tohoku-Section Joint Convention Record of Institutes of Electrical and Information Engineers, 2013, p. 1A02.

[23] P. Du, R. Weber, P. Luszczek, S. Tomov, G. Peterson, and J. Dongarra, "From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming," Parallel Computing, vol. 38, no. 8, August 2012, pp. 391–407.

[24] S. Moore, "Refactoring and automated performance tuning of computational chemistry application codes," in Simulation Conference (WSC), Proceedings of the 2012 Winter, December 2012, pp. 1–9.

[25] J. M. Levesque, R. Sankaran, and R. Grout, "Hybridizing S3D into an exascale application using OpenACC: An approach for moving to multi-petaflops and beyond," in 2012 International Conference on High Performance Computing, Networking, Storage and Analysis (SC), November 2012, pp. 1–11.

[26] M. Sugawara, K. Komatsu, S. Hirasawa, H. Takizawa, and H. Kobayashi, "Implementation and evaluation of the nanopowder growth simulation with OpenACC," The Special Interest Group Technical Reports of IPSJ, Tech. Rep. 10, 2012.

[27] G. S. Murthy, M. Ravishankar, M. M. Baskaran, and P. Sadayappan., "Optimal loop unrolling for GPGPU programs," in 2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS), April 2010, pp. 1–11.

[28] B. Bao and C. Ding, "Defensive loop tiling for shared cache," in 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), February 2013, pp. 1–11.

[29] C. Liao and D. Quinlan, A ROSE-Based End-to-End Empirical Tuning System for Whole Applications, Lawrence Livermore National Laboratory, Livermore, CA 94550, July 2013. [Online]. Available: http://rosecompiler.org/autoTuning.pdf [retrieved: 2015.02.09]

[30] CAPS, "HMPP codelet generator directives." 2008. [Online]. Available: https://www.olcf.ornl.gov/wp-content/uploads/2012/02/HMPPWorkbench-3.0_HMPPCG_Directives_ReferenceManual.pdf [retrieved: 2015.02.09]

[31] T. Soga et al., "Performance evaluation of NEC SX-9 using real science and engineering applications," in Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, ser. SC'09. New York, USA: ACM, 2009, pp. 28:1–28:12.

 ISBN: 978-1-61208-394-0