

# Expressing system-awareness as code transformations for performance portability across diverse HPC systems

[Extended Abstract]

Hiroyuki Takizawa  
Graduate School of  
Information Sciences  
Tohoku University  
6-6-01 Aramaki-aza-aoba  
Sendai, Japan

takizawa@cc.tohoku.ac.jp

Shoichi Hirasawa  
Graduate School of  
Information Sciences  
Tohoku University  
6-6-01 Aramaki-aza-aoba  
Sendai, Japan

hirasawa@sc.cc.tohoku.ac.jp

Kazuhiko Komatsu  
Cyberscience Center  
Tohoku University  
6-3 Aramaki-aza-aoba  
Sendai, Japan

komatsu@cc.tohoku.ac.jp

Ryusuke Egawa  
Cyberscience Center  
Tohoku University  
6-3 Aramaki-aza-aoba  
Sendai, Japan

egawa@cc.tohoku.ac.jp

Hiroaki Kobayashi  
Cyberscience Center  
Tohoku University  
6-3 Aramaki-aza-aoba  
Sendai, Japan

koba@cc.tohoku.ac.jp

## ABSTRACT

Our research project is developing a code transformation framework, Xevolver, to allow users to define their own code transformation rules and also to customize typical rules for individual applications and systems. By expressing system-awareness as code transformations, users do not need to directly modify the application code for system-aware performance optimizations. This prevents the application code from being specialized for a particular HPC system, and thereby the application can achieve a high performance portability across diverse HPC systems. This paper introduces our research activities and some case studies to discuss the strengths and limitations of our approach to separation of system-awareness from application codes. Using only standard technologies and tools, Xevolver can achieve important system-aware performance optimizations, such as loop optimizations and data layout optimizations, without complicating the original application code.

## Categories and Subject Descriptors

D.2.6 [Software Engineering]: Coding Tools and Techniques

## General Terms

Languages, Performance

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

*Workshop on Portability Among HPC Architectures for Scientific Applications* 2015 Austin, Texas USA  
Copyright 2015 HPCPort ....

## Keywords

Code transformation, performance optimizations, system-awareness

## 1. INTRODUCTION

Even today, system-aware performance optimizations are imperative to exploit the potential of a modern HPC system. As HPC system architectures are getting more complicated and diversified, the importance of system-aware performance optimizations will further increase in an upcoming exascale-computing era. One severe problem in system-aware performance optimizations is that, as the name implies, an HPC application code is optimized only for a particular system architecture and thus the performance may not be portable to another system. HPC application development and maintenance will become to require much more time and efforts due to the complexity and diversity of future HPC system architectures.

To tackle this challenging problem, we have started a research project to explore an effective way to separate system-awareness from an HPC application code. Our approach to the separation is to express system-awareness as code transformation by source-to-source translation. Manual code modifications are replaced with mechanical code transformations defined in external files, in order to avoid specializing an HPC application code itself for a particular system. As a result, we will be able to improve the portability of an HPC application code across different systems in terms of both function and performance.

One important research interest in this project is how code transformations can be used to replace as various code modifications as possible. Practical performance optimizations are never just to combine well-known loop transformations such as simple loop unrolling and loop tiling. Even if we use code transformation tools proposed so far, it is usually unavoidable to manually modify an HPC application code. Generally, code modifications for system-aware performance

optimizations are likely to be application-specific. As far as we know, there is no established way to abstract those code modifications in a systematic fashion.

Our only consolation is that there are repetitive patterns in such code modifications. Code modifications often seen in practice are roughly classified into several patterns, even though there could be numerous variations of each pattern. Therefore, we are developing a code transformation framework, Xevolver, to allow users to define their own code transformation rules and also to customize typical rules for individual applications and systems [8][9].

In this paper, we introduce our research activities in the Xevolver project. This paper first describes our empirical study to investigate repetitive code modification patterns in practical performance optimizations. Then, this paper reviews the Xevolver code transformation framework to abstract those code modifications as code transformations in a reusable and customizable way. Our case studies are also presented to discuss the strengths and limitations of our approach.

## 2. REPETITIVE CODE MODIFICATION PATTERNS IN REAL-WORLD APPLICATIONS

We empirically know that there are repetitive patterns in code modifications for system-aware performance optimizations. One reason of the repetitiveness is that HPC application codes tend to be themselves repetitive. This is because a typical scientific simulation code repeats processing a computational grid in various ways, and its loop nests for accessing the grid could have similar structures. A lot of similar loop nests in an application often need to be modified similarly for several reasons, resulting in repetitive code modification patterns.

An HPC application code needs to be written in such a way as to exploit the architectural features of a modern HPC system. For example, one of our target applications, Numerical Turbine [4], is already optimized for NEC SX-series vector supercomputers installed in Tohoku University Cyberscience Center [12]. The lengths of its inner-most loops are maximized for vectorization by using the loop-level parallelism. On the other hand, it is not always best for graphics processing units (GPUs) to exploit the inner-most loop parallelism. As a result, loop interchange is required to re-optimize the loop nests for GPUs. A difficulty in loop interchange arises when there exists a data dependency between the two nested loops to be interchanged. In the case where the data dependency can be removed in some way, there are repetitive patterns in code modifications for the removal. Indeed, Numerical Turbine has 44 similar loop nests to simulate various physical phenomena on one computational grid, and almost the same code modifications are required to optimize all of those loop nests by loop interchange. If a programmer wants to migrate the application to a GPU, she/he needs to modify all of the 44 loop nests in almost the same way, which will be boring and time-consuming.

As well as loop optimizations, data layout optimizations play an important role in system-aware performance optimizations. Since an HPC application code is usually written in a low-level programming language such as C or Fortran, a data structure is represented using low-level language constructs and hence bound to a particular data layout. If we convert such a data structure to another one, we need to

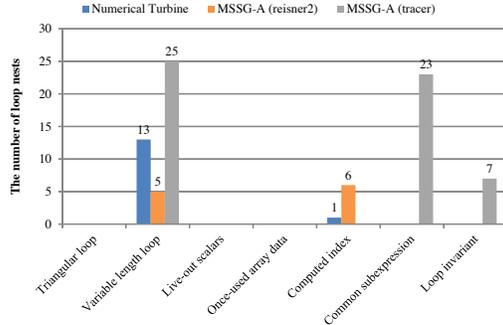


Figure 1: The number of loop nests with OpenACC compiler taboos.

replace not only its structure declaration but also every reference to the data structure. A large number of variable references could be scattered over a whole application code, and it is a labor-intensive and error-prone task to replace all of them consistently and correctly.

In addition to considering the system architecture, an HPC application code also needs to be compiler-friendly so that a compiler can fully optimize the code and thereby extract the system performance. In other words, such a code would be modified so as to avoid violating various taboos, which potentially inhibit a compiler to optimize the code. For example, there are several taboos in adapting a code to an OpenACC platform [11], which assumes OpenACC directives in an application code to use GPUs as accelerators. Major taboos include Triangular loop, Variable length loop, Live-out scalars, Once-used array data, Computed index, Common subexpression, and Loop invariant.

As a preliminary study, we have investigated how often each kind of taboos appear in real-world applications. Our target applications, Numerical Turbine [4] and MSSG [7], have been developed for NEC SX-series vector supercomputers for a long time. Hence, it is obvious that the codes have been optimized under assumption of using NEC SX compilers, and written without considering taboos of the OpenACC compiler. Figure 1 shows the investigation results to discuss how many loop nests in their main kernels violate each kind of taboos. The results clearly indicate that a kind of taboos could repeatedly appear within a kernel. Similar code modification patterns are often required to remove those taboos [13]. Therefore, this preliminary study supports our assumption that repetitive code modification patterns are often seen in system-aware performance optimizations of real-world applications.

In many cases, expert knowledge and experiences about performance optimizations are documented as programming tips and guidelines, which are not machine-readable. By reading the documents, hence, other programmers manually optimize their codes. Such code modification is likely to be repetitive routine work, and it is hence labor-intensive and boring. However, since advanced intellectual work is also essential to apply the knowledge and experiences to a specific code, it is difficult for any compilers and other programming tools to fully automate system-aware performance optimizations. Therefore, our project intends not to fully automate

the optimizations, but to represent expert knowledge and experiences in a reusable and customizable way to reduce programmers’ burden of repetitive code modifications.

We have been accumulating expert knowledge and experiences as a database called an HPC refactoring catalog [1]. One difficulty is that the code modifications are likely to be application-specific and/or system-specific, resulting in low performance portability and/or maintainability. To overcome the difficulty, Xevolver allows users to define their own code transformation rules in such a way that the rules are reusable and customizable for individual applications and systems. By replacing repetitive manual code modifications with a smaller number of custom code transformations, Xevolver can achieve high performance portability without major modifications of the original code.

### 3. XEVOLVER CODE TRANSFORMATION FRAMEWORK

In our project, the Xevolver framework is designed to enable users to define their own code transformations. The framework needs to be flexible and expressive enough to abstract a code modification as a user-defined code transformation. To this end, at the lowest level, a code transformation is defined by a transformation rule of an abstract syntax tree (AST). An AST is the internal representation of code structures used in compilers. AST transformation is what compilers do for code transformation. In our project, many case studies using real-world applications have demonstrated that Xevolver can provide various code transformations frequently required in practice.

One important difference between Xevolver and compilers is that Xevolver offers an interface to users to concretely direct code transformations, while compilers have been developed to automate decision making of such code transformations based on code analysis. The code analysis of a compiler is sometimes unable to work as expected by users, unless the code is written to be friendly to the compiler. By manually directing code transformations based on expert knowledge and experiences, Xevolver can prevent an application code from being specific to system architectures, compilers, libraries, and so on, resulting in a high performance portability.

Xevolver has so far been developed on top of the ROSE compiler infrastructure [5]. Xevolver provides the interconversion between an ROSE AST and its XML representation. Xevolver converts a ROSE AST to an XML representation of the AST, called an XML AST. Then, an XML AST is exposed to users. After some AST transformations, the transformed XML AST is again converted back to a ROSE AST so that ROSE can unparse it to a C or Fortran code. Xevolver can easily collaborate with ROSE, which provides various features of code analyses and transformations to implement custom code transformation programs in C++. Using ROSE’s features, users do not need to reimplement the same features from scratch for Xevolver.

Xevolver currently adopts XSLT (eXtensible Stylesheet Language Transformation) [3] for describing AST transformations. XSLT is a standard specification to describe XML data conversion in an XML format, and thus an AST and its transformation rules are both written in XML. As a result, Xevolver enables us to express various code transformations by using only standard XML technologies and tools.

This feature is important for representing and accumulating expert knowledge and experiences about performance optimizations in a widely-available and future-proof way.

Furthermore, we are now developing several higher-level interfaces to describe user-defined code transformations in order to explore an effective way to represent expert knowledge and experiences. For example, a JSON interface is developed to define a custom directive and associate it with a combination of predefined code transformation rules [10]. Thus, if an application-specific code transformation can be expressed by a combination of basic transformation rules such as loop unrolling and loop interchange, users just write a simple JSON file to combine those rules and associate the combination with a user-defined directive. If an expert user writes a code transformation rule in XSLT, we can increase the number of predefined rules available for the interface. In this way, we expect that expert knowledge and experiences are accumulated in a reusable and machine-readable fashion.

Another high-level interface under active development is named Xevtgen [6]. The purpose of Xevtgen is to allow users to define a custom code transformation rule by just writing two versions of a code, which are the original code and its transformed code. Using the two versions of a code, Xevtgen will automatically generate a code transformation rule written in XSLT. Therefore, users will no longer need to manually write any XSLT rules. This work is still ongoing and will be further described in our future work.

### 4. CASE STUDIES AND DISCUSSIONS

This paper shows several case studies of using XSLT to express AST transformations for practical system-aware performance optimizations. The purpose of these case studies is to explore appropriate coverage of using user-defined code transformations on system-aware performance optimizations.

To demonstrate that system-awareness is certainly separated from an application code, an existing application already optimized for one system is migrated to another system without major code modifications. Since we have a lot of real-world applications optimized for the SX vector supercomputer, those application codes are migrated to GPU systems, for which GPU-aware performance optimizations play a key role to achieve high performance. The system specifications used for the case studies are listed in Table 1.

As mentioned in Section 2, Numerical Turbine has 44 similar loop nests that have to be transformed to achieve a high performance on an OpenACC platform. The code transformation uses loop interchange so that GPUs efficiently execute the code. The GPU-aware code modification of Numerical Turbine makes inner-most loops shorter and hence remarkably degrades the performance when the application runs on the SX system, for which the code has originally been optimized. It is obvious that GPU-aware performance optimization reduces the SX performance. This is a clear instance of why we need to separate system-awareness from an application code.

In this case study, the code modification is defined as an application-specific code transformation rule, which is applicable to all of the 44 loop nests for the GPU-aware performance optimization. As a result, the application code can be transformed to another version of the code optimized for GPUs. As the GPU-awareness is expressed as a code transformation rule in an external file, the application code

Table 1: System specifications.

CPU	Model	Intel Core i7-930
	No. cores	4
	Clock [GHz]	2.8
	L3 cache [Mbytes]	8
	Memory BW [GB/s]	25.6
CPU	Model	NEC SX-9
	No. cores	1
	Clock [GHz]	3.2 (vector unit) 1.6 (scalar unit)
	Cache [Kbytes]	256
	Memory BW [GB/s]	256
GPU	Model	NVidia Tesla C2070
	No. CUDA cores	448
	Clock [GHz]	1.0
	Memory BW [GB/s]	144
GPU	Model	NVidia Tesla K20
	No. CUDA cores	2496
	Clock [GHz]	0.7
	Memory BW [GB/s]	208

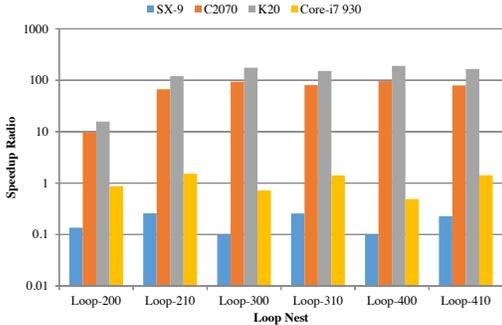


Figure 2: The performance impact of the code transformations.

hardly changes from the original one, and just a few custom directives are inserted to the code for annotation.

As a result of code transformation as the GPU-aware performance optimization, we have the original and transformed versions of each loop nest that are optimized for the SX and GPU systems, respectively. In this case study, both of the two versions are executed on each system and the performance difference between the them is evaluated. Figure 2 shows the evaluation results to discuss the performance impact of the code transformation. The vertical axis indicates the speedup ratio of the transformed version to the original one. The horizontal axis shows time-consuming six loop nests of Numerical Turbine. The GPU performance of the original version is extremely low, because the OpenACC compiler cannot exploit the loop parallelism at all. On the other hand, the OpenACC compiler can exploit the parallelism of the transformed loop nests. Therefore, the code transformation can significantly improve the GPU performance.

Since the original version is optimized for the SX system, the transformed version running on the SX system is slower

than the original one, and thus the speedup ratio is less than one. However, this is not a problem in practical use because the code transformation can be disabled when the application is executed on the SX system. The application code can be transformed to another version only when it is compiled for the GPU system. In this way, Xevolver can achieve performance portability across totally-different HPC systems.

Another case study uses the Himeno benchmark and the 27-point stencil kernel in EPCC OpenACC benchmark suite [2] to demonstrate that Xevolver can be used for data layout optimizations. Since those benchmarks already use Structure-of-Array (SoA) for structured data, the data are rewritten in an Array-of-Structure (AoS) manner, and the performance difference caused by changing the data representation is evaluated for discussions on the performance impact of data layout optimizations.

Code transformation rules to convert AoS data to SoA data are written in XSLT for each benchmark<sup>1</sup>. As mentioned in Section 2, such a transformation rule can significantly reduce the programmers’ burdens for data layout optimizations, because AoS-to-SoA data conversion is a highly-repetitive and error-prone task.

Figure 3 shows the performance impact of AoS-to-SoA data conversion. Code i7-930 and Tesla K20 in Table 1 are used as the CPU and the GPU, respectively. In the case of executing the 27-point stencil kernel on the CPU, the performance improvement is small because the array size is less than the last-level cache capacity. On the other hand, the data size of the Himeno benchmark is larger than the cache capacity, the AoS-to-SoA conversion significantly improves the CPU performance. The performance improvement of the GPU is more significant than that of the CPU, because the effective memory bandwidth of the GPU is sensitive to memory access patterns. Since the AoS-to-SoA conversion allows more memory accesses to be coalesced, the GPU can achieve a better memory bandwidth and hence a higher performance.

As demonstrated above, Xevolver can provide user-defined AST transformations for system-aware performance optimizations, loop optimizations and data layout optimizations, which play a key role to exploit the performance of a modern HPC system. However, we do not claim that Xevolver itself is a perfect solution to the performance portability problem. Xevolver obviously has some weak points while it also has its unique strong points, e.g., allowing users to define custom AST transformations for a particular case.

One weak point of Xevolver is that it does not help users define “correct” transformation rules, because it is designed to replace manual code modifications. Hence, as in manual code modifications, users are still required to be responsible for the correctness of the optimized code. Correctness-checking methods such as unit tests and assertions might be needed whenever a user-defined AST transformation as well

<sup>1</sup>In the case of the 27-point stencil kernel, discrete arrays are used instead of SoA. This is because we found that the GPU performance with discrete arrays is better than that with SoA. Use of discrete arrays is more compiler-friendly than use of SoA in the case of this particular code. For simplicity of explanation, we call AoS-to-SoA conversion even if the data structure is converted to discrete arrays. Use of discrete arrays instead of SoA does not make the transformation difficult at all because SoA simply means a data structure of packing discrete arrays.

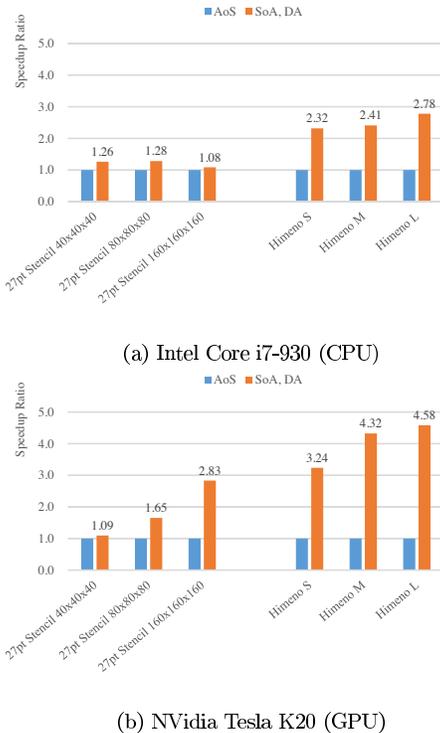


Figure 3: The speedup ratio by data layout optimizations

as manual code modification is applied to a code. Whereas, conventional compiler-based tools provide a more reliable code transformation based on advanced code analyses, which have intensively been studied for a long time. Thus, if there is such an existing solution to abstract a certain code modification, it is better to simply use the tool. In reality, however, we yet have to manually modify the code because it is common that there is no tool to do the same modification as we need. For example, in the above case studies, Xevolver is needed for adapting a code to the OpenACC compiler, and the compiler is responsible for correctly translating the code to a GPU program. It would be interesting to clarify the appropriate usage and coverage of user-defined AST transformations for expressing system-awareness. Therefore, an appropriate division of labor between Xevolver and other tools will further be discussed in our future work based on more case studies.

If the system-awareness is separated from an application code by appropriately using user-defined AST transformations, the application would be able to achieve a high performance portability across different system architectures, system scales, and system generations. Since the life of a practical HPC application code is usually longer than the life of an HPC system, the performance portability across system generations will be more important for long-term maintenance of HPC applications.

## 5. CONCLUSIONS

We often see repetitive code modification patterns in system-aware performance optimizations. To abstract such code modifications as code transformations, the Xevolver code transformation framework has been developed. Unlike compilers, Xevolver allows users to direct concrete transformations. In Xevolver, both an AST and its transformation rules are written in XML. Therefore, users can transform an AST using only standard XML technologies and tools. That is, expert knowledge and experiences are represented in a machine-readable, customizable, and future-proof way.

Our case studies clearly demonstrate that Xevolver can provide important system-aware performance optimizations, which are necessary to exploit the performance of a modern HPC system. Since the system-awareness is represented as code transformations and separated from an application code, Xevolver improves the performance without major modifications of the original code. This is helpful to achieve high performance portability across diverse HPC systems because each system can use its own code transformation rules for system-aware performance optimizations.

XSLT has a high expressiveness to describe XML data conversion, and hence AST transformation. However, most HPC programmers are not very familiar with XML technologies. Therefore, we are developing higher-level interfaces, such as Xevtgen [6], to describe code transformation rules more easily without manually writing any XSLT rules. This will appear in our future work.

## Acknowledgments

This research was partially supported by JST CREST “An Evolutionary Approach to Construction of a Software Development Environment for Massively-Parallel Heterogeneous Systems” and Grant-in-Aid for Scientific Research(B) #25280041. The authors would like to thank team members of the CREST project for fruitful discussions on the design and development of the Xevolver framework.

In addition, this research was partially supported by “Joint Usage/Research Center for Interdisciplinary Large-scale Information Infrastructures” and “High Performance Computing Infrastructure” in Japan for using the computing resources installed in Cyberscience Center, Tohoku University.

## 6. REFERENCES

- [1] R. Egawa, K. Komatsu, and H. Kobayashi. Designing an hpc refactoring catalog toward the exa-scale computing era. *Sustained Simulation Performance 2014*, 2014.
- [2] N. Johnson. EPCC OpenACC benchmark suite, 2013. <https://www.epcc.ed.ac.uk/research/computing/performancecharacterisation-and-benchmarking/epcc-openaccbenchmark-suite>.
- [3] M. Kay. *XSLT 2.0 and XPath 2.0 Programmer’s Reference (Programmer to Programmer)*. Wrox Press Ltd., 4 edition, 2008.
- [4] S. Miyake, S. Yamamoto, Y. Sasao, K. Momma, T. Miyawaki, and H. Ooyama. Unsteady flow effect on nonequilibrium condensation in 3-D low pressure steam turbine stages. In *ASME Turbo Expo 2013*, 2013.
- [5] D. Quinlan. ROSE: Compiler support for object-oriented frameworks. *Parallel Processing Letters*, 10(02n03):215–226, 2000.

- [6] R. Suda, S. Hirasawa, and H. Takizawa. User-defined source-to-source code transformation tools using Xevolver. Presentation at International Workshop on Legacy HPC Application Migration (LHAM2014), 2014.
- [7] K. Takahashi, A. Azami, Y. Tochiara, Y. Kubo, K. Itakura, K. Goto, K. Kataumi, H. Takahara, Y. Isobe, S. Okura, H. Fuchigami, J.-i. Yamamoto, T. Takei, Y. Tsuda, and K. Watanabe. World-highest resolution global atmospheric model and its performance on the earth simulator. In State of the Practice Reports, SC '11, pages 21:1–21:12, 2011.
- [8] H. Takizawa, S. Hirasawa, Y. Hayashi, R. Egawa, and H. Kobayashi. Xevolver: An XML-based code translation framework for supporting HPC application migration. In IEEE International Conference on High Performance Computing (HiPC), 2014.
- [9] H. Takizawa, S. Hirasawa, and H. Kobayashi. Xevolver: An XML-based programming framework for software evolution. Poster presentation at Supercomputing 2013 (SC13), 2013.
- [10] H. Takizawa, D. Sato, S. Hirasawa, and H. Kobayashi. A high-level interface of xevolver for composing loop transformations. to appear in Sustained Simulation Performance 2015, 2015.
- [11] The Portland Group PGI. 11 tips for maximizing performance with OpenACC directives in Fortran, 2013. [https://www.pgroup.com/resources/openacc\\_tips\\_fortran.htm](https://www.pgroup.com/resources/openacc_tips_fortran.htm).
- [12] Tohoku University Cyberscience Center. Supercomputing system. <http://www.cc.tohoku.ac.jp/>.
- [13] C. Wang, S. Hirasawa, H. Takizawa, and H. Kobayashi. Identification and elimination of platform-specific code smells in high performance computing applications. *International Journal of Networking and Computing*, 5(1):180–199, 2015.