A Use Case of a Code Transformation Rule Generator for Data Layout Optimization

Hiroyuki Takizawa, Takeshi Yamada, Shoichi Hirasawa and Reiji Suda

Abstract Xevolver is a code transformation framework for users to define their own code transformation rules. In the framework, an abstract syntax tree (AST) of an application code is written in an XML format, and its transformation rules are expressed in the XSLT format, which is a standard XML format to describe XML data conversion; an AST and its transformation rules are both written in XML. Since it is too low-level for standard users to manually write XSLT rules, Xevtgen is now being developed as a tool to generate such rules from simple code description. In Xevtgen, users basically write just two code patterns, the original and transformed code patterns. Then, Xevtgen automatically generates a transformation rule that transforms the original code pattern to the transformed one. The generated rule is written in XSLT, and hence usable by other tools of the Xevolver framework. This article shows a use case of using Xevtgen for data layout optimization, and discusses the benefits of using the tool.

1 Introduction

When data are stored in a memory space, the layout of data often needs to be optimized so as to make a better use of memory hierarchy and architectural features. Today, such data layout optimization is critically important to achieve high performance on a modern high-performance computing (HPC) system, because the system

H. Takizawa (✉) · T. Yamada · S. Hirasawa
Graduate School of Information Sciences, Tohoku University, Sendai, Japan
e-mail: takizawa@tohoku.ac.jp

T. Yamada
e-mail: tyamada@sc.cc.tohoku.ac.jp

S. Hirasawa
e-mail: hirasawa@sc.cc.tohoku.ac.jp

R. Suda
Graduate School of Information Science and Technology,
The University of Tokyo, Tokyo, Japan
e-mail: reiji@is.s.u-tokyo.ac.jp

M.M. Resch et al. (eds.), *Sustained Simulation Performance 2016*,
DOI 10.1007/978-3-319-46735-1_3

performance is very sensitive to memory access patterns. Memory access can easily become a performance bottleneck of an HPC application.

The data layout of an application can be optimized by changing data structures used in the code. One problem is that a human-friendly, easily-understandable data representation is often different from a computer-friendly data layout. This means that, if the data layout of a code is completely optimized for computers, the code may be no longer human-friendly.

We have been developing a code transformation framework, Xevolver, so that users can define their own rules to transform an application code [1, 2]. In this article, such a user-defined code transformation rule is adopted to separate the data representation in an application code from the actual data layout in a memory space. Instead of simply modifying a code for data layout optimization, the original code is usually maintained in a human-friendly way and then mechanically transformed just before the compilation so as to make the transformed code computer-friendly.

One important question is how to describe code transformation rules. A conventional way of developing such a code translator is to use compiler tools, such as ROSE [3]. Actually, at the lowest abstraction level, Xevolver allows users to describe a code transformation rule as an AST transformation rule. Since AST transformation is exactly what compilers internally do, compiler experts can implement various code transformation rules by using the framework. However, standard programmers who optimize HPC application codes are not necessarily familiar with such compiler technologies. Therefore, we are also developing several high-level tools to describe the rules more easily.

Xevtgen [4] is one of high-level tools to help users define custom code transformation rules. This article shows a use case of Xevtgen for data layout optimization, and discusses how it can help users define their own transformations.

2 Data Layout Optimization

In many cases, an HPC application code is written in a low-level programming language such as C/C++ and Fortran. In such a language, a data structure mostly corresponds to a specific data layout. In practice, thus, the data layout of an HPC application is usually altered by changing the data structure in the code.

A typical example of data layout optimization is so-called AoS-to-SoA conversion [5]. Generally, an array of structures (AoS) is likely to be human-friendly, leading to high code maintainability and readability. For example, the following C code defines an AoS data structure, in which each point is a pair of two variables, `x` and `y`.

```
struct { double x, y; } point2d[N];
```

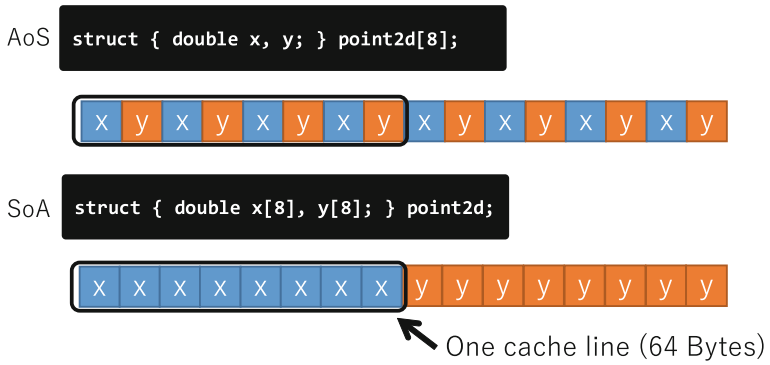


Fig. 1 Data layout in a memory space

This would be an intuitive representation of points in a 2-dimensional space. On the other hand, a structure of arrays (SoA) often leads to better memory access patterns. The following is an example of SoA, in which `x` and `y` are both arrays.

```
struct { double x[N], y[N]; } point2d;
```

Each data structure leads to a different data layout in a memory space as shown in Fig. 1. In the case of AoS, `x` and `y` appear alternately in the memory space. On the other hand, in the case of SoA, each of the two arrays organizes a continuous memory region. For example, when `x` of every point is sequentially accessed, it is obvious that SoA has a higher spacial locality of reference. As a result, SoA can potentially use cache memory more effectively. In this way, the data layout in a memory space can significantly affect the performance of an HPC application.

One severe problem is that data layout optimization needs to modify many places in an application code, and degrades the code maintainability and readability. As long as the data representation in an application code corresponds to its actual data layout in a memory space, data layout optimization results in drastic code modification. Moreover, it could reduce the performance portability across different systems because different systems potentially prefer different data layouts. Programmers may need to maintain multiple versions of one application code, e.g., using many `#ifdef`'s, especially if the application needs to be performance-portable. This approach likely makes the code unmaintainable.

One idea to solve this problem is to use code transformation. For example, various compiler tools are available to transform a code instead of directly modifying the code. However, transformation for data layout optimization is generally specific to a particular application code. This is because such a transformation rule usually depends on the definition of the data structure. In general, it is not affordable to use a compiler tool to develop a custom code translator for individual applications. Accordingly, we need an easier way to define a custom code transformation rule.

3 Code Transformation Rule Generation with Xevtgen

Suppose that a legacy code written in Fortran uses a human-friendly data structure. Then, this article discusses how the code should be converted to another version of the code that uses a computer-friendly data structure. Thus, the purpose of this conversion is to develop a code using human-friendly data representation and execute it using computer-friendly data layout. To this end, we transform the code instead of modifying it. That is, the code is transformed just before the compilation, and then the transformed version is passed to the compiler. As a result, application developers maintain only the original version.

One difficulty is that this conversion needs application-specific code transformations in many cases. The Xevolver framework has been developed to allow standard users to implement such an application-specific code transformation. First, Xevolver internally converts an application code to its AST, represents the AST in an XML data format, and then exposes it to users. The users can apply any transformations to the exposed AST. After the AST transformation, Xevolver converts the transformed AST to its corresponding code. At the lowest abstraction level, Xevolver internally uses XSLT [6] to express an AST transformation rule, because XSLT is a standard XML data format to describe XML data conversion. It should be noted that an AST and its transformation rules are both written in XML, i.e., text data. In comparison with other data formats, it would be easy to quickly develop a simple tool to generate and/or process text data for application-specific purposes. Therefore, we have been developing various tools for the Xevolver framework and discussing the practicality and superiority of using user-defined code transformation for optimizing an HPC application code.

Xevtgen [4] is one of the tools to generate XSLT rules for AST transformation. In the case of using Xevtgen, users do not need to consider their code transformation rules at an AST level. Instead, they write two code patterns, the original and transformed code patterns. Then, Xevtgen automatically generates an XSLT rule of custom code transformation that transforms the original code pattern to the transformed one.

Figure 2 shows an example of "dummy" Fortran codes to express custom code transformations. Such a dummy code is used by Xevtgen to generate XSLT rules for AST transformation. In Fig. 2, the original and transformed code patterns are written using `!$xev tgen src` and `!$xev tgen dst` directives, respectively. In the dummy code, some special variables can be used to express a code pattern. For example, a variable, `idx`, defined in Line 27 matches any expression, and hence is used in the rule to indicate that an array index can be any expression. If such a variable appears in both of the original and transformed code patterns, the corresponding part of the code pattern is copied from the original code to the transformed one. Similarly, a variable, `cmp`, matches any name so that a component of `aos_t` is translated to its corresponding component of `soa_t`.

Reading a dummy Fortran code with special directives as in Fig. 2, Xevtgen generates XSLT rules that implement code transformations expressed in the dummy

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!! A SIMPLE EXAMPLE OF DUMMY FORTRAN CODE                              !!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
program dl

!! Rule 1
!! Replace the type definition of AoS data structure
!$xev tgen src begin
  type aos_t
        real:: x
        real:: y
  end type aos_t

  type(aos_t) aosdata(100)
!$xev end tgen src
!$xev tgen dst begin
  type soa_t
        real:: x(100)
        real:: y(100)
  end type soa_t

  type(soa_t) soadata
!$xev end tgen dst

!! Rule 2
!! Change the way of accessing the data structure
!$xev tgen var(idx) exp
!$xev tgen var(x) name
!$xev tgen trans exp src('aosdata(idx)%x') dst('soadata%x(idx)')

end program dl
```

Fig. 2 A dummy Fortran code written for Xevtgen

code. This is the most important feature of Xevtgen. In the case of using Xevtgen, users do not need any knowledge about AST transformation; they can write a dummy code to express custom code transformations if they know Fortran programming and several `!$xev tgen` directives. Using Xevtgen, users can easily and quickly define a code transformation dedicated to a particular application code. In practice, such an application-specific code transformation rule does not need to have high generality if necessary rules can easily be defined for each application code. As a result, an application code can remain human-friendly from the viewpoint of application programmers, and user-defined code transformations are applied to the code just before the compilation so that the transformed code becomes computer-friendly and/or compiler-friendly.

Generally, two kinds of code transformations are required for AoS-to-SoA conversion. One is called a *type definition transformation rule* that replaces the type definition of an AoS data structure to that of an SoA data structure. An example of a type definition transformation rule is Rule 1 in Fig. 2. The other is called a *variable reference transformation rule* that changes the way of accessing the data structure. An example of a variable reference transformation rule is Rule 2 in Fig. 2.

In many cases, a type definition transformation rule is explicitly defined by users as in Fig. 2. It often looks like simple textual replacement. Since a type definition

appears only once in an application code, simple textual replacement is usually an easy and appropriate way to achieve type definition transformation.

On the other hand, a variable reference transformation rule often needs to be expressed using generalized code patterns rather than concrete texts. This is because a data structure could have various components. If simple text replacement is adopted for this kind of transformation, the rule will be very wordy. To make matters worse, since those variable references could be scattered over a whole application code, it is a tedious and error-prone task to manually change all of them for using a different data structure. If Xevtgen is used and there is a general code pattern that matches all the variable references, a variable reference transformation rule could be simple as shown in Fig. 2. Once such a simple rule is defined, all the variable references are mechanically transformed based on the rule.

In the dummy code in Fig. 2, the part of a directive surrounded by back quotes, such as `` `aosdata(idx)%x` ``, is a Fortran expression. The expression must be valid when Xevtgen translates the dummy code to XSLT rules. Hence, a Tgen variable, `x`, is defined to have the same variable name as a component of `aos_t` so that `` `aosdata(idx)%x` `` is a valid expression. Since `x` is a Tgen variable as well as a component of `aos_t`, it matches any name when the generated XSLT rule is applied to a Fortran code. As a result, the generated rule replaces not only `aos(i,j,k)%x` but also `aos(i,j,k)%y` at the AoS-to-SoA conversion.

4 Discussions

In this article, Xevtgen is used for AoS-to-SoA conversion to discuss how Xevtgen helps users define application-specific code transformations. In the following evaluation, the classic static memory allocation version of the Himeno benckmark [7] written in Fortran77 is first modified by hand so as to use AoS data representation, and then a dummy Fortran code for Xevtgen is written to generate XSLT rules for AoS-to-SoA conversion. In addition, the performance difference between the AoS version and the SoA version is also shown to discuss how important data layout optimization is for modern computing systems.

Figure 3 shows a dummy Fortran code used by Xevtgen for AoS-to-SoA conversion in this use case.

For variable reference transformation, the rule in Fig. 3 can be expressed as a simple code of only four lines because there is a clear pattern in the transformation. The code pattern of accessing the original AoS data structure is mechanically replaced with the code pattern of accessing arrays. Since variable references are scattered over a whole application code, it is significantly helpful if all of them are transformed based on a certain transformation rule, especially in the case of optimizing the data layout of a large-scale practical application code. Even in a relatively-small code of the Himeno benchmark, 61 variable references need to be modified for changing the data structure.

```
program dl_himeno
  !! Rule 1
  !! Replace the type definition of AoS data structure
  !$xev tgen src begin
  type aos_t
     sequence
     real::p
     real::a(4)
     real::b(3)
     real::c(3)
     real::bnd
     real::wrk1
     real::wrk2
  end type aos_t
  common /aos/ aosdata
  type(aos_t) aosdata(mimax,mjmax,mkmax)
  !$xev end tgen src
  !$xev tgen dst begin
  common /pres/ p(mimax,mjmax,mkmax)
  common /mtrx/ a(mimax,mjmax,mkmax,4), &
       b(mimax,mjmax,mkmax,3),c(mimax,mjmax,mkmax,3)
  common /bound/ bnd(mimax,mjmax,mkmax)
  common /work/ wrk1(mimax,mjmax,mkmax),wrk2(mimax,mjmax,mkmax)
  !$xev end tgen dst

  !! Rule 2
  !! Change the way of accessing the data structure
  !$xev tgen var(i,j,k,l) exp
  !$xev tgen var(p,a) name
  !$xev tgen trans exp src(`aosdata(i,j,k)%p`) dst(`p(i,j,k)`)
  !$xev tgen trans exp src(`aosdata(i,j,k)%a(l)`) dst(`a(i,j,k,l)`)
end program dl_himeno
```

Fig. 3 A dummy Fortran code for AoS-to-SoA convesion of the Himeno benchmark

In Fig. 3, the type definition transformation rule consists of 21 code lines. One may consider that the type definition transformation rule seems redundant because both of the original and transformed code versions are explicitly written. However, the type definition of a transformed data structure must be explicitly given by users anyway, considering the correspondence between the original and transformed types. Therefore, we believe that this is an effective and intuitive way to describe a type definition transformation rule.

In this use case, 60 code lines are in total transformed by the rules in Fig. 3. In case of converting AoS to SoA, a programmer traditionally needs to modify all of the code lines by hand. This is a tedious and error-prone task. On the other hand, in the case of using Xevtgen, user-defined code transformation can automate the task if the rule is properly defined. Moreover, the rule is reusable and/or easily customizable for another code. Accordingly, Xevtgen dummy codes can be used for accumulating and sharing expert knowledge and experiences, which can be reused by other programmers and also for other application codes.

In this article, the performance of the transformed code is compared with that of the original code using the two systems listed in Table 1. One system is the NEC SX-ACE vector computing system [8] installed at Tohoku University Cyberscience Center, and the other is a commodity PC. The former has a 10x higher memory bandwidth than

the latter. Since the performance of the Himeno benchmark is usually limited by the memory bandwidth, SX-ACE can achieve a higher sustained performance than the PC if the loop is properly vectorized.

Figure 4 shows the performance impact of the data layout optimization. As shown in the figure, AoS-to-SoA conversion significantly improves the performance of each system. The performance improvement of SX-ACE is especially remarkable, because regular memory access to array elements, which are arranged in a continuous region, is more efficient than stride memory access to AoS data elements. In this use case, it is observed that the AoS-to-SoA conversion can significantly reduce memory bank conflicts and improve the memory access efficiency. This kind of performance optimization is very important to achieve high performance on a modern computing system, in which the memory access can easily become a performance bottleneck. The expressive ability of Xevtgen is high enough to express transformation rules of this important performance optimization.

It is important that different systems potentially require different data layouts. As a result, data layout optimization is likely to be system-specific and/or compiler-specific. Moreover, loop optimization might also require data layout optimization because it changes the order of accessing data elements. For example, even if SoA had a higher spacial locality of reference than AoS before optimizing a loop, SoA does not necessarily have a higher locality after the optimization. Data layout optimization often specializes an application code for a particular processor, compiler, kernel loop, and so on. Even in such a case, our Xevolver approach [1] allows the original code to remain human-friendly because system-specific and/or compiler-specific code transformations can be defined separately from the original code. Namely, system-specific and/or compiler-specific information is separated from the code.

Xevtgen enables users to achieve the separation much more easily than the original Xevolver approach. In this use case, the XSLT rules generated by Xevtgen consist of 293 lines in total. In the original Xevolver framework, users are supposed to write such XSLT rules by themselves, as reported in [9]. Writing those XSLT rules requires not only a fair amount of writing effort but also special knowledge about both XML and compilers, because users have to learn how to write XSLT rules for AST transformation. On the other hand, Xevtgen allows users to briefly express their own code transformation rules without requiring special knowledge about XML, XSLT,

Table 1 System configurations

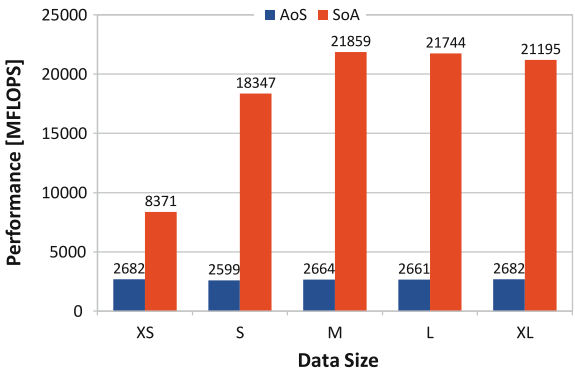
	NEC SX-ACE	Commodity PC
Processor	SX-ACE processor	Intel Core i7 930
Peak performance	256 Gflop/s	44 Gflop/s
Memory capacity	64 GB	32 GB
Memory bandwidth	256 GB/s	25.6 GB/s
Operating system	SUPER-UX	Linux 2.6.32
Compiler	sxf90 Rev. 520	gfortran 4.4.7
Compiler options	-P auto -C hopt	-O3

and compilers. As a result, they can easily improve the performance portability of their application codes across different systems.

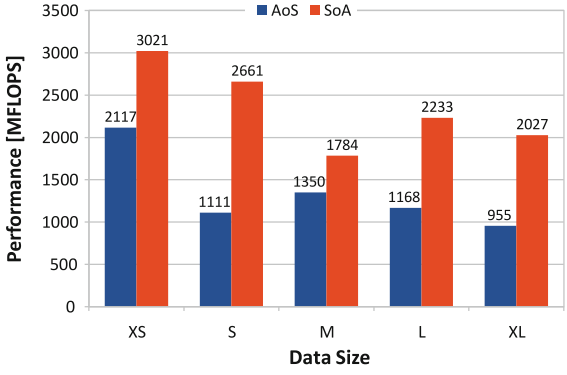
5 Conclusions

This article explains a code transformation rule generator, Xevtgen, for user-defined code transformations dedicated to each application code. As discussed in this article, Xevtgen allows standard users to define their own code transformations much more easily than conventional compiler-based approaches, because the users no longer need to consider code transformation rules at an AST level. They can generate code transformation rules if they know Fortran programming and several special directives.

The use case described in this article shows that Xevtgen can express code transformations required for data layout optimization, which is one of the most important code optimization techniques to exploit the performance of a modern computing



(a) SX-ACE.



(b) Commodity PC.

Fig. 4 Performance impacts of data layout optimization

system. In terms of the number of code lines, XSLT rules generated by Xevtgen are longer than a dummy Fortran code input to Xevtgen. Moreover, standard Fortran programmers would be able to describe such a dummy Fortran code if they learn how to use several special directives used by Xevtgen. In comparison with the original Xevolver approach [1] of directly writing XSLT rules by hand, the Xevtgen approach offers a much easier way of defining a practical code transformation rule.

Acknowledgements 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) 16H02822. The authors would like to thank all team members of the CREST project, especially Profs. Egawa, Takahashi, and Komatsu, for fruitful discussions on the design and development of the Xevolver framework.

References

1. Takizawa, H., Hirasawa, S., Hayashi, Y., Egawa, R., Kobayashi, H.: Xevolver: an XML-based code translation framework for supporting HPC application migration. In: IEEE International Conference on High Performance Computing (HiPC) (2014)
2. The Xevolver Project: JST CREST "an evolutionary approach to construction of a software development environment for massively-parallel heterogeneous systems". http://xev.arch.is.tohoku.ac.jp/
3. Quinlan, D.: ROSE: Compiler support for object-oriented frameworks. Parallel Process. Lett. **10**(02n03), 215–226 (2000)
4. Suda, R., Takizawa, H., Hirasawa, S.: Xevtgen: Fortran code transformer generator for high performance scientific codes. In: The Third International Symposium on Computing and Networking, pp. 528–534 (2015)
5. Sung, I.J., Liu, G.D., Hwu, W.M.W.: DL: a data layout transformation system for heterogeneous computing. In: Innovative Parallel Computing (InPar), pp. 1–11 (2012)
6. Kay, M.: XSLT 2.0 and XPath 2.0 Programmer's Reference (Programmer to Programmer), 4 edn. Wrox Press Ltd. (2008)
7. Himeno benchmark. http://accc.riken.jp/en/supercom/himenobmt/
8. Momose, S., Hagiwara, T., Isobe, Y., Takahara, H.: The brand-new vector supercomputer, SX-ACE. In: International Supercomputing Conference, pp. 199–214. Springer (2014)
9. Yamada, T., Hirasawa, S., Takizawa, H., Kobayashi, H.: A case study of user-defined code transformations for data layout optimizations. In: The Third International Symposium on Computing and Networking, pp. 535–541 (2015)