Making a Legacy Code Auto-tunable without Messing It Up

Hiroyuki Takizawa Daichi Sato Shoichi Hirasawa Hiroaki Kobayashi
Graduate School of Information Sciences, Tohoku University
6-6-01 Aramaki-aza-aoba, Aoba, Sendai 980-8579, Japan
{takziawa@, da175@sc.cc, hirasawa@sc.cc, koba@}tohoku.ac.jp

ABSTRACT
Since computing platforms are diverging, it is becoming more important to make a legacy code “auto-tunable” so that the code is capable of adapting to various platforms including future-generation systems. However, such a code is likely to be considerably complex and hard-to-maintain because it is supposed to change its code structure according to several parameters. In this work, we discuss how to use auto-tuning technologies without overcomplicating the code itself. This work focuses on user-defined code transformation that can be defined separately from an application code. Then, this work demonstrates that a framework named Xevolver allows users to easily define code transformation rules for transforming a particular code to its auto-tunable version. As a result, application developers can usually maintain the original version, and the code is transformed just before auto-tuning. If code transformation rules are properly defined by experts, the application developers can benefit from auto-tuning technologies without considering the complex auto-tunable code generated by the transformation.

1. INTRODUCTION
Automatic performance tuning, or auto-tuning, has recently been attracting intense attention [1] because it is no longer affordable to optimize one application code for all of diverse computing platforms by hand. As the diversity of high-performance computing (HPC) system architectures is further increasing, the importance of auto-tuning technologies will also increase more and more towards an extreme-scale computing era.

One difficulty in practical use of auto-tuning is that most of existing application codes are developed without assuming any auto-tuning technologies, and thus need to be modified so as to be auto-tunable. That is, for auto-tuning, an application code must be capable of changing its performance by tuning some parameters. The task of making an existing code auto-tunable could be challenging especially if the code is a so-called legacy code, which has been maintained by various programmers for a long time.

One problem with a legacy code is that such a code is usually written in a relatively low-level programming language. Since a lot of various programmers have been involved in the development, no one has a holistic understanding of the code, and hence there is no unified way to alter performance-sensitive code fragments scattered over the whole code. Another problem is that legacy code developers are likely to avoid major code modifications as much as possible, because a legacy code has been used for a long time and proven to produce correct results. Because of those problems, the task of making a code auto-tunable is error-prone and labor-intensive.

This work focuses on user-defined code transformation to make legacy codes auto-tunable. A code transformation rule defined separately from an application code is applied to the code just before auto-tuning. Application developers do not need to care about the transformed code if the rules are properly defined by experts, called performance engineers. Accordingly, user-defined code transformation enables an appropriate division of labor between application developers and performance engineers.

2. THE XEVOLVER FRAMEWORK
We have been developing a user-defined code transformation framework, Xevolver, so that users can define their own code transformations [3]. At the lowest abstraction level, Xevolver internally uses eXtensible Markup Language (XML) to express an Abstract Syntax Tree (AST) of a code, because XML is a well-established way of expressing any tree data such as ASTs and also there is a standard data format, eXtensible Stylesheet Language Transformations (XSLT), to express XML data conversion. If necessary, hence, users can consider code transformations at an AST level.

We are also developing high-level interfaces such as Xevtgen [2] for standard users to easily define transformation rules for special demands of individual applications. In the case of using Xevtgen, users basically write two versions of a code, the original and transformed versions. Then, Xevtgen can automatically generate rules that can transform the original version to the transformed one. Therefore, users can define new transformation rules without special knowledge about XML, XSLT, and compiler technologies such as ASTs.
To make a legacy code auto-tunable, user-defined code transformations are often demanded especially if data layout optimization is needed. A code transformation rule for optimizing a data structure usually depends on the definition of the data structure, and thus is likely specific to a particular application code. Such a transformation rule cannot be defined by conventional code transformation approaches of composing predefined rules. Therefore, this work demonstrates that user-defined code transformation is helpful to introduce an auto-tuning capability to a legacy code that does not use high-level constructs for neither loops nor data structures.

3. AUTO-TUNING WITH USER-DEFINED CODE TRANSFORMATIONS

An auto-tuning framework, OpenTuner [1], is a powerful tool to efficiently explore a huge parameter space and quickly find an appropriate parameter configuration. However, OpenTuner assumes that the target code being tuned is already auto-tunable; performance-sensitive parts of the code have been parameterized and thus a high performance is achieved by just finding optimal (or suboptimal) parameters. In this work, therefore, Xevolver is used for making an existing code auto-tunable so that OpenTuner can be easily applicable, while the original code is kept almost unchanged.

To discuss the benefits of combining the two frameworks, this work tunes the Himeno benchmark by making decisions for the following three tuning points. One is to select either loop blocking or loop collapse to optimize the kernel loop. For loop blocking, the block size is also determined. Another is to determine either discrete arrays or an array of structures for data to be accessed within the kernel loop. The last one is to decide 426 compiler options that can best extract the performance of the code.

Supporting more tuning parameters usually results in further complicating the code. In the original version of the Himeno benchmark, the number of code lines of its kernel loop is only 10. In addition to the original kernel loop, we consider two variants of the loop, blocked and collapsed loops. In this work, these loops are further transformed for supporting two kinds of data structures. As a result, six variants including the original loop are written in the same code, and exclusively used based on a so-called \texttt{ifdef} approach, which really degrades the code maintainability.

If we do not use any code transformation or generation, one problem is the growth in code size. In our case, the total number of code lines of the six variants is 65 when the C preprocessor is used for conditional compilation with \texttt{ifdefs}. Another problem is that the original loop body is duplicated for each variant. We have to edit all of the duplicated bodies if we change the computation of the loop.

In the case of using Xevolver, application developers can keep the original code almost unchanged. Even though the other five variants are necessary for auto-tuning in this evaluation, the variants are mechanically generated from the original code just before auto-tuning. The transformation rules consist of 100 lines in total. Even in the case of this relatively-small code, i.e., the Himeno benchmark, the number of code lines to describe the rules is less than 155 code lines that are changed or added by the rules. Since a practical scientific simulation code often contains a lot of similar loop nests [3], one transformation rule might be reusable for many loop nests. In such a case, the rules will be much shorter than codes affected by the rules.

Although Xevolver can easily generate a lot of code variants with hiding the code complexity, the parameter search space being explored during the auto-tuning process becomes larger with the numbers of variants and their parameters. By combining this code transformation approach with OpenTuner, we can efficiently explore such a huge parameter space. Figure 1 shows that the performance quickly improves as the auto-tuning process proceeds. In our experiment, full parameter search takes 71,944 seconds, while OpenTuner can achieve almost the same performance in about 3,000 seconds on average. Application developers can enjoy this performance gain while hiding the code complexity behind code transformations defined by performance engineers. We believe that this separation of concerns is very helpful to achieve an appropriate division of labor between those two kinds of programmers.

4. CONCLUSIONS

In this work, we have shown a case study of using user-defined code transformations for making a legacy code auto-tunable. For making it auto-tunable, application-specific coding techniques are often required. By using custom code transformations for individual applications, application developers do not need to directly modify those codes themselves. As a result, they can enjoy the benefit of auto-tuning without degrading the code maintainability. In our future work, we will integrate auto-tuning and code transformation more tightly so that they can collaborate more easily and effectively.

5. REFERENCES