Portability of Vectorization-aware Performance Tuning Expertise across System Generations

Shunpei Sugawara*, Yoichi Shimomura†, Ryusuke Egawa‡‡, Hiroyuki Takizawa†*

* Graduate School of Information Sciences, Tohoku University, shunpei@hpc.is.tohoku.ac.jp
† Cyberscience Center, Tohoku University, {shimomura32,takizawa}@tohoku.ac.jp
‡ Tokyo Denki University, egawa@mail.dendai.ac.jp
Background

Software automatic tuning (Auto-tuning)
- To reduce the effort of code optimization, auto-tuning adjusts various parameters (performance knobs) that affect performance. ex.) Block size, Loop length, etc.

Portability of performance knobs
- Is a performance knob for old systems effective also for new systems?
  - Or, new performance knobs are needed?
  → The application code would be overcomplicated if it has too many performance knobs.
This Work

Objectives

- To discuss the portability of effective performance tuning techniques across system generations
- To explore a way of cataloging portable techniques in a future-proof way

Contributions

- Empirically showing how differently a performance tuning technique affects the performance of each system generation
- Discussing how a performance tuning technique should be expressed and applied to an application.
Related Work

- **Xevolver code transformation framework** [1]
  - System-neutral *Fortran* codes are translated into system-specific codes according to user-defined code translation rules

![Diagram]

Application code

Predefined and/or user-defined annotations

Original Ver. + Optimized Ver.

Xevolver

A

Application code optimized for A

B

Application code optimized for B

Translation rule defined by user

Related Work

- **HPC refactoring catalog** [2]
  - 31 cases of performance tuning expertise
    - Target systems: NEC SX-9 and SX-ACE
    - Code examples: mostly in Fortran

- **Is it still useful for new systems?**
  - **NEC SX-Aurora TSUBASA (SX-AT)** [3]
    - The latest generation of SX-series systems
    - Totally different system architecture and software stack
      - C/C++ as well as Fortran used

---

Translating Fortran to C

Why C?

- C language is getting more popular in HPC
- We are now developing Xevolver\(^1\) for C
  - A code translation framework for maintaining performance portability

28 out of 31 examples are translated

- 3 code examples cannot be translated because these examples contain:
  - Fortran’s built-in functions that do not exist in C
  - library functions that do not have C interface

Xevolver for C: translation rule

- User-defined code translation rule
  - `xev_expr n`: any variable, expression
  - `xev_stmt stmt`: any statement
  - `xev_stmt_src, dst`: indicates the code before and after the translation
  - `xev_expr_replace`: translate a specific expression to a specified form

```
#include "xev_defs.h"

int i,j;
int xev_expr n;
int xev_stmt* stmt;

int main()
{
  xev_stmt_src("label1");
  {
    for(i=0;i<10;i++){
      for(j=0;j<10;j++){
        stmt;
      }
    }
  }
  xev_stmt_dst("label1");
  {
    for(j=0;j<10;j++){
      for(i=0;i<10;i++){
        stmt;
      }
    }
  }
  xev_expr_replace(pow(n,2),(n)*(n));
}
```
Xevolver for C: before and after translation

Applying the translate rule from the previous slide

```c
int main()
{
    int x, a, b, c, i, j;

    for(i=0; i<10; i++)
    {
        for(j=0; j<10; j++)
        {
            c = a * i + b * j;
        }
    }
    x = pow(c, 2);
}
```

```c
int main()
{
    int x, a, b, c, i, j;

    for(j=0; j<10; j++)
    {
        for(i=0; i<10; i++)
        {
            c = a * i + b * j;
        }
    }
    x = c * c;
}
```
Evaluation setup

- 28 Examples of HPC refactoring catalog written in C

System specifications
- SX series have vector processor VE and x86 processor VH to run Linux OS

Compiler
- ncc -3.2.0(-O2,-O4)
- ncc -2.5.1(-O2,-O4)
- gcc-4.8.5
- NEC LLVM-IR Vectorizer

Metric

\[ R = \frac{T_b}{T_a} \]

\( T_b \): the execution time before the transformation

\( T_a \): the execution time after the transformation
Evaluation: between different systems

- Performance improves in ncc -O2: 20 out of 28 techniques
- There is little difference between them in terms of the effects of optimization techniques
Evaluation: Effects of Compiler options

- Performance improves in:
  - ncc -O4: 12 out of 28 techniques

- This is probably due to the more powerful compiler optimizations in -O4
Evaluation: Performance difference in compiler

- In some cases, the performance is unchanged or decreased
- In LLVM-clang compiler, the performance degrades in more cases

![Graph showing performance differences for different case IDs and compiler types](image-url)
Evaluation: Vectorization-aware loop optimization cases

- In Cases 5 and 11, compiler finds that the loops could be executed faster with scalar instructions.
- In Case 6, the average vector length increases and the number of vector instructions decreases.
**Evaluation**: compiling the performance tuning cases with gcc

- Most of vectorization-aware loop optimization techniques are not effective for the x86 processor
Conclusions

- This work empirically demonstrates
  - The performance tuning techniques for SX-9 and SX-ACE are still effective even for SX-AT with the C compiler
    - With a new compiler, some techniques are no longer effective
  - Performance tuning techniques for one system may not be effective for another system.

- Conclusions
  - Performance tuning expertise in the past is useful even for
    - Future system generations
    - New compilers
    - Other programming languages
  - But advances in compiler technology may outdate some.
  - The importance of separating system-specific optimization from application codes (= Xevolver’s approach)