

# Code Coverage, Performance Approximation and Automatic Recognition of Idioms in Scientific Applications

Jiahua He &  
Allan E. Snively

CSE Dept. & SDSC  
Univ. of California, San Diego  
9500 Gilman Drive,  
La Jolla, CA 92093  
j2he@cs.ucsd.edu  
allans@sdsc.edu

Rob F. Van der Wijngaart

Software Solutions Group,  
Intel Corporation  
3600 Juliette Lane,  
Santa Clara, CA 95054  
rob.f.van.der.wijngaart@intel.com

Michael A. Frumkin

Google Corporation  
1600 Amphitheatre Parkway,  
Mountain View, CA 94043  
frumkin@google.com

## ABSTRACT

Basic data flow patterns which we call **idioms**, such as stream, transpose, reduction, random access and stencil, are common in scientific numerical applications. We hypothesize that a small number of idioms can cover most programming constructs that dominate the execution time of scientific codes and can be used to approximate the application performance. In this paper, we start with a manual analysis of code coverage on the NAS Parallel Benchmark (NPB) and find that five idioms suffice to cover 100% of the NPB codes. We then compare the performance of our idiom benchmarks and their corresponding instances in different NPB codes on two different platforms and find that they differ by about 30%. To check the hypotheses with real applications further, we propose an automatic idioms recognition method, implement the method basing on the open source compiler Open64, and verify the prototype system with the previous manual analysis results.

## Categories and Subject Descriptors

C.4 [Computer Systems Organization]: PERFORMANCE OF SYSTEMS—*Modeling techniques*

## General Terms

Measurement, Performance

## Keywords

Scientific applications, performance analysis, application requirements, idioms recognition

## 1. INTRODUCTION

In scientific applications, a number of similar constructs, design patterns, and data flows can be found. The performance analysis community has long realized this and utilized the common programming constructs and typical application kernels as benchmarks. But how to quantify the representativeness of benchmarks remains a hard problem.

Usually the selection of benchmarks is to some extent subjective and decided by human sense. Our research find that identification of the common constructs or data flow patterns, such as stream, transpose, reduction, random access and stencil, in real applications, along with their coverage coefficients, would allow us to use these constructs as performance proxies for real applications quantitatively and allow us to derive requirements for new platforms based on the analysis of a few simple constructs.

In this paper, we call these similar constructs **idioms**. Formally, idioms are primitive program components, which each capture a pattern of computation and communication over arrays or sub-arrays common in applications. Some questions about the idioms are: how many idioms do we need to cover most application codes? Can these idioms approximate the performance of real applications? And can we find out these idioms from application codes automatically? To answer the above questions, we will start with a manual analysis of code coverage on application codes and then compare the performance of our idiom benchmarks and their corresponding instances in application codes. Finally, we will develop an automatic recognition tool and verify it with the previous manual analysis results. To this end, we use the NAS Parallel Benchmark (NPB) suite [2] as a case study. Without losing generality, we only consider Fortran codes in this paper.

## 2. CODE COVERAGE

In this section, we analyze the NPB suite to find out all the idioms and their **code coverage**, that is, the percentages of the static code classified as idioms (dynamic code coverage will be the next step). NPB includes five kernels (EP, MG, CG, FT, and IS) and three pseudo applications (BT, LU and SP). Since EP is not expected to be covered (fortunately, it is about input generation and not a dominant part of scientific computation) and IS is written in the C language, we only applied our analysis to the other six programs. Our analysis results show that the above five idioms suffice to cover all the assignments of the NPB programs within loops. There are two examples worthy of mention here. The subroutine *Swarztrauber()* in FT can be viewed as a “shuffle” on the function level. But our analysis proceeds on the statement level and all the assignments of the subroutine are classified as Stream. Another example is the

Sparse-Matrix-Vector multiplication (SMV) in CG, which is classified as a hybrid idiom composed of Reduction and Random. Though real application codes might be much more complicated and versatile than NPB, at least the result verified that it is feasible to use a small number of idioms to cover application codes.

### 3. PERFORMANCE APPROXIMATION

In this section, we develop a set of idiom benchmarks and try to use their measured performance to approximate the performance of idiom instances in the NPB programs. For simplicity, each idiom benchmark was written in a single typical form of the idiom. Also, in each NPB benchmark, only the performance-dominant instances are considered. Each idiom benchmark scaled across a large enough range of data set sizes to cover those of all the corresponding instances in NPB. We measured the benchmarks and the NPB codes on two different platforms: SDSC IA64 cluster and DataStar. The total average difference on IA64 cluster is 30.2% and that on DataStar is 36.8%. The performance matching between the idioms and their instances in the NPB codes proves to some extent that it is feasible to use simple idioms to approximate the performance of real applications.

### 4. AUTOMATIC RECOGNITION BY COMPILER

In this section, we are proposing a compiler-based automatic idioms recognition method using affinity relations. If variables  $A$  and  $B$  are on the left hand side (LHS) and right hand side (RHS) of an assignment, respectively, the **affinity relation** between  $A$  and  $B$  is a matrix composed of the coefficients of their indexes if they are array variables or just 0's otherwise. If a term including an index that is not linear, its coefficient is marked as *FUNC*. The method includes four stages.

First, we need to extract the loop nest structure from the compiler intermediate representation to construct the Loop Nest Graph (LNG), which is used to represent the control flow structure of the whole routine, especially the loop nest structure. Its goal is to provide context information for the following analysis such as surrounding loops, loop indexes and loop-carried dependences. The LNG we consider is a directed tree. Its root represents the whole routine and other nodes stand for natural loops or other control structures such as If-statements. There is a directed edge between two nodes if and only if the control flow construct of the parent node includes that of the child node.

With the LNG in hand, we can traverse the whole control flow structure recursively to scan all the assignments and build Affinity Relation Graphs (ARG) for them. To obtain context information easily, each ARG is hung under its corresponding control flow construct in the LNG. An ARG itself is also a directed tree graph. Its root represents the variable on LHS, which equals the whole expression on RHS, and other nodes stand for the sub-expressions (or variables on leave nodes) on RHS. There is a directed edge between two nodes if and only if the expression of the parent node includes that of the child node.

Though we have caught affinity relation between variables in an ARG, it is still not enough for idiom recognition because there are temporary variables in the middle to interfere. We need to remove these and reduce the ARG into a

Reduced Affinity relation Graph (RAG). Instead of deleting the nodes directly from the ARG, we chose to build a new graph by copying the root and the leaves from the ARG and connecting them accordingly.

Now we have a RAG for each combined assignment. We can also easily obtain dependence information from the compiler dependence graph. We then can check each RAG to see if it satisfies any idiom definitions and classify the assignment as the matched idiom. Since both affinity relation and dependence cycle are related to the surrounding loops, idioms have to be classified with respect to a specific surrounding loop.

### 5. PROTOTYPE IMPLEMENTATION AND EXPERIMENT RESULTS

The Open64 compiler [1] is an industrial-strength compiler and contains an advanced and complete optimization framework, including scalar optimization (WOPT), loop nest optimization (LNO), inter procedural optimization (IPA) and code generation (CG). Its intermediate language called WHIRL was designed to have five levels (Very High, High, Mid, Low and Very Low) to fit the different requirements of different analyses and optimizations. Our prototype was implemented in the LNO phase of the High WHIRL level. In this phase, the high level control flow constructs such as loops and If-statements are preserved and array structures are kept, which aids our implementation, particularly the construction of the LNG. The LNO phase also provides the facilities for dependence analysis, which is essential for our implementation. To verify the prototype system, we applied it to the NPB and compared the results with those of our manual analysis. The highest error is 10.2%. Our results prove that automatic idiom recognition is feasible.

### 6. CONCLUSIONS

We are interested in if small idioms can cover most scientific codes and approximate the performance of real applications. We first tested these two hypotheses by hand with the NPB suite and found that the above five idioms suffice to cover all the NPB codes and the performance difference between the idioms and their corresponding instances in the NPB is of about 30%. These results prove to some extent that it is feasible to use a small number of idioms to cover some application codes and approximate their performance. To verify our hypotheses with real applications, we proposed an automatic idioms recognition method and implemented the method, based on the open source compiler Open64. Comparing the automatic analysis results with the previous manual ones showed that it is feasible to recognize the idioms automatically.

### 7. ACKNOWLEDGEMENTS

We thank the Intel<sup>®</sup> Internship Program and University Cooperation Program for supporting this work, Dr. Lars Jonsson for supervising the project.

### 8. REFERENCES

- [1] Open64 Compiler home page.  
<http://www.open64.net/>.
- [2] D. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, and M. Yarrow. The NAS parallel benchmarks 2.0. Technical Report NAS-95-020, NASA, Dec. 1995.