Automatic Recognition of Performance Idioms in Scientific Applications

Jiahua He*, Allan E. Snavely*, Rob F. Van der Wijngaart† and Michael A. Frumkin‡
* *San Diego Supercomputer Center (SDSC), University of California, San Diego*
† *Software Solutions Group, Intel Corporation*
‡ *Google Corporation*
Emails: jiahua@gmail.com, allans@sdsc.edu, rob.f.van.der.wijngaart@intel.com, frumkin@google.com

***Abstract*—Basic data flow patterns that we call performance 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. To check these hypotheses, we proposed an automatic idioms recognition method and implemented the method, based on the open source compiler Open64. With the NAS Parallel Benchmark (NPB) as a case study, the prototype system is about 90% accurate compared with idiom classification by a human expert. Our results showed that the above five idioms suffice to cover 100% of the six NPB codes (MG, CG, FT, BT, SP and LU). We also compared the performance of our idiom benchmarks with their corresponding instances in the NPB codes on two different platforms with different methods. The approximation accuracy is up to 96.6%. The contribution is to show that a small set of idioms can cover more complex codes, that idioms can be recognized automatically, and that suitably defined idioms may approximate application performance.**

***Keywords*-Performance evaluation; Compiler; Idioms; Scientific applications**

I. Introduction

In scientific applications, space, time, local and long-range interactions are represented by abstract programming constructs. Despite many different implementations, there are fundamental similarities in these abstractions: space is usually modeled by structured or unstructured grids, time is often modeled by the outermost iteration loop (time step), and interactions are modeled by systems of equations that are solved by iterative explicit or implicit methods.

As a result, a number of similar constructs, design patterns, and data flows can be found in various scientific applications [18], [24], [22]. Extraction, measurement and analysis of these representative constructs can result in a good understanding of the application performance and suggesting how best to map the applications to computer architectures. 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 found 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 application requirements for new platforms based on the analysis of a few simple constructs.

Understanding these program constructs helps not only new hardware design not also performance optimizations. For example, reduction is one of the program constructs well studied by the compiler communities. Quite a few compilers [25], [30], [19] implemented advanced reduction recognition for program optimization. Furthermore, hybrid hardware is emerging with different components designed for different program constructs. RoadRunner [13] is the first petaflops supercomputer with hybrid hardware: Opteron ® cores plus Cell ® processors. GPU computing is becoming more and more important. The first place of the current Top500 list [8] is GPU machine, Tianhe-1A. How to recognize suitable program constructs for GPU execution automatically will be critical for the usability and efficiency of these GPU machines. FPGA machines will have even more diversities. One example is Convey ® HC-1 [14], which provides a handful application-specific co-processors called personalities for Intel ® processors. To explore the potential of this kind of hardware needs automatic recognition of different program constructs (personalities) and mapping them to corresponding hardware components. In one of our un-published works, we were able to recognize the Gather/Scatter constructs in real applications and send it to the Convey FPGA accelerator thus speeding as much as 20x.

On the road to exascale supercomputing, performance is not the only concern. In fact, power consumption is becoming the limit for scalability. To solve the problem, our colleague Professor Andrew Chien has proposed a new paradigm called 10x10 [1]. The basic idea is to integrate 10 different specialized cores onto one chip and use the best matched ones for specific program constructs. Our recognition technique can be used to recognize these program constructs and map them to corresponding cores.

In this paper, we call these similar constructs **performance idioms** or, for simplicity, just **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? These are the questions explored in this paper. To this end, we design and develop a static analysis tool to recognize idioms automatically, and then verify the tool by manual analysis. We also try to compare the performance of our idiom benchmarks with their corresponding instances in application codes. The NAS Parallel Benchmark (NPB) [12] suite will be used as a case study. We limited ourselves to Fortran codes in this paper. Extending to other languages will be one of our future works.

The rest of the paper is organized as follows. In the next section, we will define five well-studied idioms, that is, Stream, Transpose, Random access, Reduction and Stencil. Then in section III, we will propose a compiler-based method to recognize these idioms within real application codes automatically and present an implementation based on the open source compiler Open64. The experimental results of the prototype system applied to the NPB will be described in section IV. Also the code coverage and performance approximation results of NPB will be presented. Section V includes some related work in the field. Conclusions and future work will be given in the last section.

II. Definitions of Five Idioms

Idioms represent basic operations of applications. Systems used for scientific computations should deliver high performance for all, or at least the dominant part of, the idioms. The complete list, or even just a dominant list of idioms, has not been identified yet. However, some idioms such as Stream, Transpose, Random access, Reduction and Stencil are already well studied in the community. In this section, we formally define these five idioms by the concept of affinity relation and dependence.

We can write an array variable assignment of array B to array A inside a loop nest using pointer arithmetic:

$$A + V \cdot I = B + W \cdot I,$$

where I is the vector of loop variables $(i, j, k \ldots)$ and V and W are vectors expressing affinity. The matrix $(V, W)^T$ is called the **affinity relation** between A and B. In another word, if variables A and B are on the left hand side (LHS) and the 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 is not linear, its coefficient is marked as $FUNC$. For example, the assignment $a(i, j) = b(c(j), i)$ can be transferred to its linearized form $a(i + n * j) = b(c(j) + m * i)$, where n and m are the sizes of the leading dimensions of a and b, respectively. Then the affinity relation between a and b can be represented by:

$$\begin{pmatrix} 1 & n \\ m & FUNC \end{pmatrix}$$

where the first row is for a and the second is for b. The first column is for loop variable i and the second one is for j. In general, an affinity relation A looks like:

$$\begin{pmatrix} a_{11} & a_{12} & \ldots \\ a_{21} & a_{22} & \ldots \end{pmatrix}$$

where the first row is for the LHS variable and the second one is for the RHS variable.

By **dependence**, we mean data dependence, that is, a constraint between statements to guarantee the order in which data are produced and consumed. Dependence information can be easily derived from a compiler's intermediate representation, such as a dependence graph [9], [28]. A dependence cycle is a directed cycle in the dependence graph.

With the above two definitions, we can define the five idioms as follows. One thing worthy of mention is that every definition here is with respect to a specific surrounding loop. We will talk about more details in section III-D.

Definition 1 (Stream): An assignment is classified as Stream if

- for each affinity relation A between LHS and an array variable on RHS, there do not exist two loop indexes i and j such that $a_{1i} < a_{2i}$ but $a_{1j} > a_{2j}$;
- there is no FUNC element in any affinity relation;
- it is not involved in a dependence cycle.

As follows is the triad version of the Stream benchmark designed by McCalpin [7]. If its data set size is big enough, a Stream idiom will consume the full bandwidth between the processor and the main memory.

```
do i=1, m
  a(i) = b(i) + const * c(i)
end do
```

Definition 2 (Transpose): An assignment is classified as Transpose if

- for each affinity relation A between LHS and an array variable on RHS, there exist two loop indexes i and j such that $a_{1i} < a_{2i}$ but $a_{1j} > a_{2j}$.

Here is a typical example of the Transpose idiom. It reads a matrix by rows and writes it by columns, which will challenge the strided access performance of the memory system.

```
do i=1, m
  do j=1, n
    a(j,i) = b(i,j)
  end do
end do
```

Definition 3 (Random Access): An assignment is classified as Random access if

- for each affinity relation A between LHS and an array variable on RHS, there exist at least one element marked as $FUNC$ (we understand that, in general, FUNC **could** represent a simple access pattern, but we found the assumption of random useful for performance classification).

Here is an example of Random Access idiom, which reads unpredictable memory locations and writes them into contiguous memory locations. Its performance is limited by memory latency.

```
do i=1, m
  a(i) = b(c(i))
end do.
```

Definition 4 (Reduction): An assignment is classified as Reduction if

- the LHS variable appears on the RHS with the same subscript if any;
- for each affinity relation A between LHS and an array variable on RHS, if $a_{1i} \neq 0$, there must be $a_{2i} \neq 0$;
- the first level operators of RHS are associative;
- it is involved in dependence cycles.

A typical one-dimensional Reduction idiom looks like as follows, which reduces an array by an associative operation. Its performance can be optimized by vectorization.

```
do i=1, m
  s = s + a[i]
end do
```

Definition 5 (Stencil): An assignment is classified as Stencil if

- for each affinity relation A between LHS and an array variable on RHS, there do not exist two loop indexes i and j such that $a_{1i} < a_{2i}$ but $a_{1j} > a_{2j}$;
- it is involved in dependence cycles.

Below is a typical one-dimensional Stencil idiom. Each update of an array element depends on the values of its neighbors. It tests the performance of instruction scheduling.

```
do i=1, m
  a(i) = a(i-1) + a(i+1)
end do
```

Sometimes more than one idiom definitions are satisfied. Then the assignment is classified as a hybrid idiom. For example, the Sparse-Matrix-Vector multiplication (SMV) as follows is a hybrid idiom composed of Reduction and Random.

```
do j=1, lastrow-firstrow+1
  sum = 0.d0
  do k=rowstr(j), rowstr(j+1)-1
    sum = sum + a(k)*p(colidx(k))
  enddo
  q(j) = sum
enddo
```

III. Automatic Recognition by Compiler

In the previous section, we have defined five well-studied idioms. We are wondering if it is feasible to use such a small number of idioms to completely or mostly represent certain scientific application codes and if these idioms are useful as performance proxies of real applications. In this section, we are proposing a compiler-based static analysis method to extract idioms from real application codes automatically. A prototype implementation will also be described in the last sub-section.

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**). And then traverse the loop nest to scan all the assignments and extract their affinity relations to construct Affinity Relation Graph (**ARG**). By removing temporary variables, the Reduced Affinity relation Graph (**RAG**) will be built. Finally, idioms are recognized by matching the RAG with idiom definitions. The details of these four stages will be introduced in the rest of the section. At the same time, we will go through the whole analysis process of the example Fortran routine shown in Figure 1.

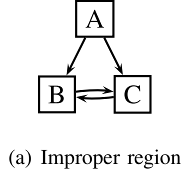
```
subroutine foo
integer i, j
do j = 1, 100
  do i = 1, 100
    tmp = a(i,j) - b(i)
    sum(i) = sum(i) + tmp*tmp
  enddo
enddo
return
end
```

Figure 1. A complete example of Fortran routine

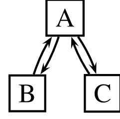
A. LNG: Loop Nest Graph

The Loop Nest Graph (LNG) 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. Here we do not consider improper regions, which are multiple-entry strongly connected components, or loops sharing the same header. Figure 2(a) shows an example Control Flow Graph (CFG) [9], [28] of improper regions. And an example CFG of loops with the same header is shown in Figure 2(b).

The LNG we consider is a directed tree. Its root represents the whole routine and other nodes stand for natural loops



(a) Improper region



(b) Loops with the same header

Figure 2. Example CFGs of improper region and loops with the same header

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. Usually, the LNG can be easily built from the compiler intermediate representation such as Control Flow Graph (CFG) or its intermediate language if it is designed to have the control flow structure embedded like the Open64 High WHIRL that we will introduce in the next section. Figure 3 shows the LNG of the example in Figure 1.

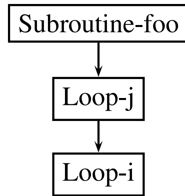


Figure 3. LNG of the example in Figure 1

B. ARG: Affinity Relation Graph

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 leaf 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. The affinity relation matrix rows are distributed across the corresponding variable nodes. Since data might flow from one assignment to another by temporary variables, we adopted a method similar to Static Single Assignment (SSA) [17] to connect these assignments together to be a so-called **combined assignment**. Specifically, a version number is attached to each variable. If it is a scalar variable, its number is incremented only while it is on LHS. Being an array variable, its number is incremented every time it appears in an assignment in order to keep the subscript information for each instance. We do not consider data flows across borders of control flow constructs and so we do not add phi-nodes [17].

Figure 4 shows the ARG of the example in Figure 1. Here the extensions of variable names stand for version numbers. Since we only increment the version number of a scalar variable when it is on LHS, all three instances of the variable tmp have the same version number and they are represented by a single node in the ARG. As a result, the two assignments are connected together and we will see that they are finally recognized as a single idiom instance. In contrast, the instances of the same array variable sum have different version numbers and they are represented by different nodes in the ARG. In fact, we use array variables as the borders of combined assignments.

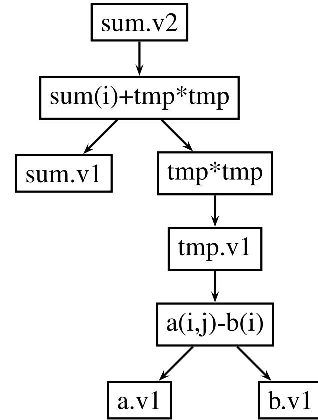


Figure 4. ARG of the example in Figure 1

C. RAG: Reduced Affinity relation Graph

Though we have caught affinity relation between variables in the 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). A RAG is still a directed tree graph. 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. By keeping the ARG intact, we can turn back to the original graph in case we need more information. The affinity relation matrix rows are still distributed across the variable nodes. Figure 5 shows the RAG of the example in Figure 1. Now we can easily see that the LHS of the combined assignment is $sum.v2$ and the RHS includes $sum.v1$, $a.v1$ and $b.v1$. In the next part,

we will analyze the affinity relations between $sum.v2$ and the RHS variables to match the idiom patterns.

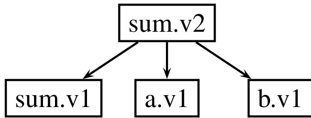


Figure 5. RAG of the example in Figure 1

D. Idioms Recognition

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 given in the section II and classify the assignment as the matched idiom. Since both affinity relations and dependence cycles are related to the surrounding loops, an idiom has to be classified with respect to a specific surrounding loop. To this end, we adopted an algorithm similar to Allen and Kennedy's vectorization algorithm [23] to classify idioms from the outer most loop to the inner ones. For a given loop, we only consider the dependence cycles whose maximum levels equal the current loop. Here **level** means the index of the leftmost non-equal element of the dependence direction vector [23]. Specifically, we only keep the dependences whose levels are higher than the current loop, that is, the loop carried dependences over the loops inside and including the current loop plus the loop independent dependences. Next we construct the maximal Strongly Connected Components (SCC) of the dependence graph. During the construction, the maximum level of each SCC is recorded. While being matched with the idiom definitions, an assignment is classified as "in dependence cycle" if and only if it belongs to an SCC whose size is more than one and the maximum level equals the current loop, or it has a self-dependence carried by the current loop.

Let us continue to analyze the example in Figure 1. As you may observe, over the j loop, there is a self-dependence for sum, from $sum.v2$ to $sum.v1$. Since the assignment also satisfies the other requirements of the Reduction definition, it is classified as Reduction with respect to the j loop. Now we come to the i loop. Since the dependence over the j loop is removed, there is no dependence cycle on the assignment at this point. The assignment also satisfies the other requirements of the Stream definition and is classified as Stream with respect to the i loop.

E. Implementation on Open64

The Open64 compiler was originally developed by SGI ® as MIPSpro, and was open-sourced later. It also benefited from the Open Research Compiler (ORC) [4] project by Intel ® and the Chinese Academy of Sciences, which focused on IA64 architecture. Open64 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.

IV. Experiment Results

In the previous section, we have described a 4-stage method and its implementation for automatic idioms recognition. Now we need to verify the accuracy of the prototype system. Also we are interested in if a small number of idioms can cover most scientific codes and approximate the performance of real applications To this end, we start with a manual analysis of code coverage on the NPB codes. And then, the result of the manual analysis is used to verify the prototype system. Finally, we try to compare the performance of our idiom benchmarks with their corresponding instances in the NPB codes.

A. The NAS Parallel Benchmark

In this section, we use the NAS Parallel Benchmark (NPB) suite [12] as a case study. NPB is a set of benchmarks designed by NASA Advanced Supercomputing division (NAS) to evaluate the performance of supercomputers. The benchmarks are derived from Computational Fluid Dynamics (CFD) applications. The reasons to choose NPB are two fold. First, it was originally developed in 1990's and has been well studied and widely used by the scientific computing community [33], [26], [29], [35], [16]. In a recent Berkeley technical report [11], the authors found that most of the important modern numerical methods have their corresponding NPB benchmarks. Second, the code size of NPB is reasonable for manual analysis of code coverage, whose results are necessary to verify the automatic recognition tool.

NPB includes five kernels (EP, MG, CG, FT, and IS) and three pseudo applications (BT, LU and SP). Since EP (representative of input generation) is not a numerical, array traversing, application, which we theorized our idioms may cover, and IS is written in the C language, we only applied our analysis to the other six programs.

B. Code Coverage

As mentioned above, we are interested in how many idioms are needed to cover most application codes; specifically, whether a small number of idioms, say, the above five idioms can cover most application codes. To answer the question, a good way is to analyze some typical scientific applications 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). We decided to start by analyzing the NPB codes to prove the feasibility. The manual analytical results presented here will also act as the verification of the automatic tool later.

Table I
STATIC BREAKDOWN OF THE NPB BY IDIOMS

	MG	CG	FT	BT	SP	LU
Stream	76	43	19	501	283	390
Transpose	0	0	2	3	3	0
Random	18	3	1	0	0	0
Reduction	2	7	0	7	47	4
Stencil	2	4	0	0	45	5
SMV	0	2	0	0	0	0
Total	98	59	22	511	378	399

Our analysis results show that the above five idioms suffice to cover all the assignments of the six NPB programs within loops. Table I shows the static breakdown of the NPB by the five idioms. 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. As mentioned above, it is classified as a hybrid idiom composed of Reduction and Random. For clarity, it is still shown separately in the table. Though real application codes might be much more complicated and versatile than the NPB, our results verified that it is feasible to use a small number of idioms to cover application codes.

C. Prototype Verification

To verify the prototype system, we applied it to the NPB and compared the results with those of our manual analysis. The results are shown in Table II. The column "Number of mis-recognized assignments" shows the number of differences between the automatic and manual analyses. And the column "Relative error" shows the ratio of the above number to the total number of assignments. We can see that the highest error is 10.2%. Our results show that automatic idiom recognition is feasible.

Table II
AUTOMATIC IDIOMS RECOGNITION FOR NPB

Benchmarks	Total number of assignments	Number of mis-recognized assignments	Relative error
MG	98	2	2.0%
CG	59	6	10.2%
FT	22	2	9.1%
BT	511	5	1.0%
SP	378	21	5.6%
LU	399	13	3.3%

D. Performance Approximation

By our definition, idioms represent the basic components of scientific applications. But what can we say about their performance? Can we use the performance of idiom benchmarks to approximate the performance of idiom instances in real application codes? If yes, it will be easier to extract application requirements for new systems and allow compiler and hardware designers to focus on the performance of a few simple idioms rather than on full applications.

Table III
CONFIGURATIONS OF THE EXPERIMENT PLATFORMS

	Itanium2	Power4
Frequency	1.5 GHz	1.7 GHz
L1 Instruction Cache	16 KB	64 KB
L1 Data Cache	16 KB	32 KB
L2 Data Cache	256 KB	1.5 MB
L3 Data Cache	6MB	128 MB
Main Memory	4 GB	32 GB

To investigate the question, we developed a set of idiom benchmarks and tried 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. For example, though there are several versions of Stream idiom in McCalpin's Stream benchmark [7], we only adopt the triad version in our experiment. Also, in each NPB benchmark, only the performance-dominant instances are considered. For a specific idiom benchmark, there may be quite a few corresponding instances. We scaled the benchmark across a large enough range of data set sizes to cover those of all the corresponding instances in the NPB codes. The benchmarks and the NPB codes were measured on two different platforms: Intel ® Itanium2 [3] processor and IBM ® Power4 [5] processor. Table III shows the detailed configurations of these two platforms.

The average difference between our idiom benchmarks and their instances in the NPB on Intanium2 is 30.2% and that on Power4 is 36.8%. To make it more intuitive, here we show the comparison between the Stream benchmark and its corresponding instances in CG on the two platforms in Figure 6 and Figure 7, respectively. CG_xy's in the figures are the Stream instances in CG. As referred above, since each idiom benchmark scales across a large enough range

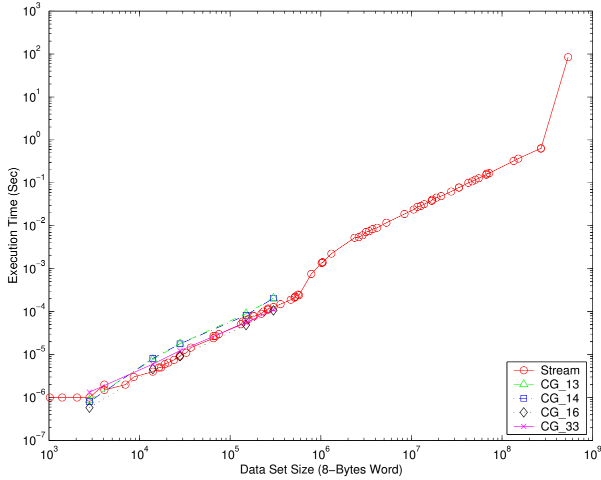


Figure 6. Execution time versus data set size of the Stream idiom benchmark and its instances in CG on Itanium2

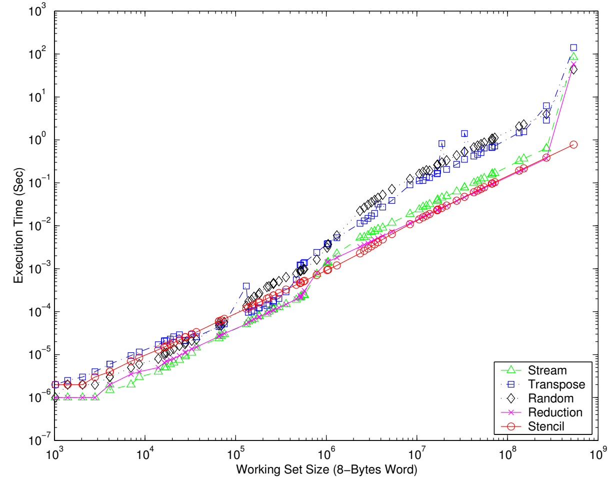


Figure 8. Execution time versus data set size of the idiom benchmarks on Itanium2

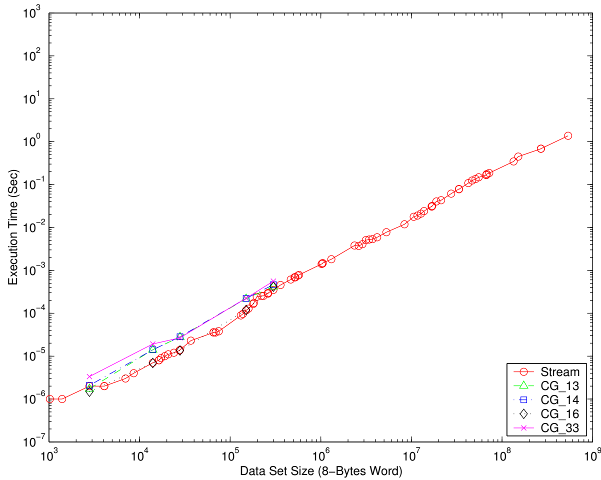


Figure 7. Execution time versus data set size of the Stream idiom benchmark and its instances in CG on Power4

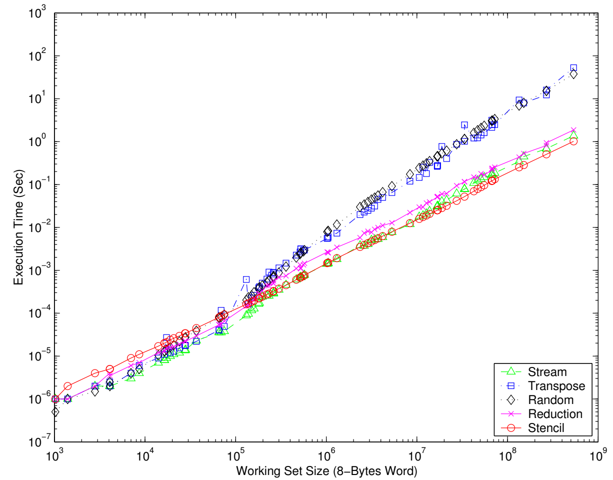


Figure 9. Execution time versus data set size of the idiom benchmarks on Power4

of data set sizes to cover those of all the corresponding instances in the NPB codes, the data set size range of a specific instance is usually smaller than that of its corresponding idiom benchmark as shown in the figures. From the curves we can see that the Stream benchmark and its corresponding instances in CG have similar trends, which also happens between other idioms and NPB programs.

We also investigated the relative matching between the idioms and their instances. The best predictor of an instance is the idiom benchmark with minimum performance difference from the instance. If the best predictor of an instance is its corresponding idiom classified by our static analysis, it is a hit. Otherwise, it is a miss. We applied the investigation to the performance-dominant instances we measured. The hit rate on Itanium2 is 79.3% and that on Power4 is 48.3%, which are not satisfying. It is because different idioms might have similar performance and interfere with each other though they have different data flow patterns. As shown in Figure 8 and Figure 9, the performance trend of Transpose is quite similar to that of Random, and the trends of the other three idioms are similar. By observing this, we grouped the idioms with similar performance together and applied the investigation again. The hit rates are much improved: 96.6% on Itanium2 and 79.3% on Power4.

In this section, we showed both the absolute and the relative performance matchings between idioms and their corresponding idioms in the NPB code. To improve the absolute performance matching, we might need to consider idiom variations, e.g. the same idiom with different memory access strides. In contrast, for relative matching, we need to group the idioms with similar performance together and have less categories. We will work more on these two directions

and explore the trade-off between them. The performance matching between the idioms and their instances in the NPB codes shows that it is feasible to use simple idioms to approximate the performance of real applications.

V. Related Work

Our project is based on many successful investigations. In this section, we would like to survey some related works in different fields.

A. Benchmarks and Application Requirements

We are not the first to propose the idea to use common programming constructs as benchmarks or application requirements. The HPC Challenge (HPCC) benchmark suite [2] was developed by the University of Tennessee, Knoxville, to measure and evaluate the performance of different components, such as processor, memory and interconnection, of high performance computers. It consists of 7 tests, ranging from low level idioms such as Stream, Parallel Matrix Transpose (PTRANS) and RandomAccess to high level kernels such as High Performance Linpack (HPL) and FFT. Though the HPCC test set has overlap with our idiom set, our idioms are all on the same low level and more general in application codes.

The Seven Dwarfs project [11] from the University of California, Berkeley tried to identify a number of basic numerical methods such as N-body method, structured and unstructured grids, MapReduce, which are important for science and engineering applications and representative for application requirements. Dwarfs are algorithmic methods whose levels are much higher than those of our idioms. They can even be the highest level algorithm of a large application. The high level of abstraction allows reasoning about the Dwarfs' behavior across a broad range of applications. But they are not easy to be recognized automatically.

B. Archetypes and Kernel Coupling

Parallel programming archetype [31] is an abstraction composed of computational structure, parallelization strategy and implied patterns of data flow and communication, proposed by researchers from California Institute of Technology for parallel program developing and performance analysis. A program developed using archetype includes two parts: archetype-specific communications and application-specific computations. To analyze its performance, we need to measure the execution time of these two parts respectively, and then build a performance model or stimulation according to the computation and communication structure implied by the archetype to calculate the whole execution time. Though the compositional performance analysis idea of the archetype is similar to ours, its granularity is much larger than us. Furthermore, it is a manual method and a programmer has to understand the program structure to decompose the program and build the performance model or simulation by hand.

The Prophesy project [34] from Texas A&M University has a similar idea of compositional performance analysis dividing a large program into small kernels to analyze. Since significant works has been done on performances of small kernels, the project focuses on kernel coupling [33], the performance relations between different kernels in a single program. By this way, performance predictions can be greatly improved over the traditional technique of just simply summing up the execution times of the individual kernels in a program. Again, comparing with our work, the granularity of kernel coupling is larger, usually on the function level, and the application decomposition (during database building) is manual. The kernel coupling idea is interesting and may be also necessary for our works if later we aim at a higher goal of performance prediction instead of just defining application requirements. However, it will be difficult if not impossible to apply kernel coupling directly on our fine-grain idioms. Some variations are probably needed.

C. Machine Idioms

Machine idioms [9] are special instruction sequences common in assembly codes. Such a sequence usually comes from a specific high-level operation (e.g., increment of a variable). Though the operation is considered by the programmer as atomic, it might be translated into a sequence of instructions on a target machine. If such an operation is common and the sequence repeats frequently, we might want to add a new instruction (e.g., instruction INC) into the ISA of the target machine and then a compiler can replace the whole sequence with a single instruction. There are quite a few works [10], [32] in the community trying to recognize high-coverage machine idioms for optimal ISA extensions or other architecture designs. Usually they also use graph matching to recognize idioms. However, these works are quite different from ours. First, a machine idiom is defined as an instruction sequence in dynamic context. It does not consider iterative behaviors over loops and sometimes is even limited in a basic block. Second, since machine idioms are specific instruction sequences, they only need simple **exact** graph matching to recognize.

D. Reduction Recognition

Reduction recognition is well studied in compiler research, especially in parallelizing compilers. The HPF compiler [25] from Rice University, the Polaris compiler [30] from University of Illinois and the SUIF compiler [19] from Stanford University are three of such systems. All of them apply dependence analysis and pattern matching methods to recognize reductions. The SUIF compiler can even recognize and parallelize interprocedural and sparse reductions. However, their goal is for optimization instead of workload characterization and performance analysis. They usually focus on only a few of the idioms that can be optimized, especially reductions. Even in these considered

idioms, they avoid some complicated forms to avoid errors or expensive optimization costs. In contrast, our method is more general and tries to catch as many idioms as possible.

VI. Conclusions and Future work

By our definition, performance idioms are the basic components of scientific applications. In this paper, we proposed an automatic idioms recognition method and implemented the method, based on the open source compiler Open64. Applied to the NAS Parallel Benchmark (NPB), the prototype system achieved an accuracy of more than 90% compared with a manual idioms classification. We also found that five idioms suffice to cover 100% of the six NPB codes (MG, CG, FT, BT, SP and LU) and the performance approximation between the idioms and their corresponding instances in the NPB codes is up to 96.6% accurate. Our automatic recognition method and prototype system enable us to find out the representative idioms of real scientific applications automatically. Our preliminary results proved to some extend that a small number of idioms can cover most scientific codes and approximate the performance of real applications.

With the verified automatic tool, we will try to prove the code coverage hypothesis by examining more and larger application codes such as CPU2006 [6]. At this point, we are still working on static code coverage. One of our next steps is to calculate the dynamic code coverage by combining the static results with the dynamic profile information. With dynamic code coverage, we then can approximate the application performance automatically and check more application codes to test our hypothesis about performance approximation. We are also working to apply the technique for performance optimization on GPU and FPGA machines. Moreover, according to our previous research on non-volatile memory [21], [15], [20], [27], performance behaviors on these new storage technologies are totally different from traditional spinning disks. As a result, similar ideas can be applied for hybrid storage systems.

Acknowledgements

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

References

[1] Andrew Chien, "Does 10x10 replace 90/10", Salishan'10. http://www.lanl.gov/orgs/hpc/salishan/index10.shtml.

[2] HPC Challenge benchmark. http://icl.cs.utk.edu/hpcc/.

[3] Itanium wikipedia page. http://en.wikipedia.org/wiki/Itanium.

[4] Open Research Compiler. http://ipf-orc.sourceforge.net/.

[5] Power4 wikipedia page. http://en.wikipedia.org/wiki/POWER4.

[6] SPEC CPU2006 home page. http://www.spec.org/cpu2006/.

[7] STREAM: Sustainable memory bandwidth in high performance computers. http://www.cs.virginia.edu/stream/.

[8] Top500 (Nov. 2010). http://www.top500.org/lists/2010/11.

[9] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers — Principles, Techniques, and Tools*. Addison Wesley, Pearson Education, Inc., 1986.

[10] G. Araujo, P. Centoducatte, M. Cortes, and R. Pannain. Code compression using operand factorization. *Proceedings of the 31th Annual International Symposium on Microarchitecture*, 1998.

[11] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec. 2006.

[12] 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.

[13] K. J. Barker, K. Davis, A. Hoisie, D. J. Kerbyson, M. Lang, S. Pakin, and J. C. Sancho. Entering the petaflop era: the architecture and performance of roadrunner. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–11, Piscataway, NJ, USA, 2008. IEEE Press.

[14] T. Brewer. Instruction Set Innovations for Convey's HC-1 Computer. *The 21st Symposium of High Performance Chips (HotChips)*, 2009.

[15] A. M. Caulfield, J. Coburn, T. Mollov, A. De, A. Akel, J. He, A. Jagatheesan, R. K. Gupta, A. Snavely, and S. Swanson. Understanding the Impact of Emerging Non-Volatile Memories on High-Performance, IO-Intensive Computing. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[16] R. Cheveresan, M. Ramsay, C. Feucht, and I. Sharapov. Characteristics of workloads used in high performance and technical computing. In *ICS '07: Proceedings of the 21st annual international conference on Supercomputing*, pages 73–82, New York, NY, USA, 2007. ACM.

[17] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, Oct. 1991.

[18] M. Frumkin. Data flow pattern analysis of scientific applications. In *Workshop on Patterns in High Performance Computing*, May 2005.

[19] M. W. Hall, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, and M. S. Lain. Interprocedural parallelization analysis in SUIF. *ACM Transactions on Programming Languages and Systems*, 27(4):662–731, Jul. 2005.

[20] J. He, J. Bennett, and A. Snavely. DASH-IO: an empirical study of flash-based IO for HPC. In *Proceedings of the 2010 TeraGrid Conference*, TG '10, pages 10:1–10:8, New York, NY, USA, 2010. ACM.

[21] J. He, A. Jagatheesan, S. Gupta, J. Bennett, and A. Snavely. DASH: a Recipe for a Flash-based Data Intensive Supercomputer. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[22] J. He, A. E. Snavely, R. F. Van der Wijngaart, and M. A. Frumkin. Code coverage, performance approximation and automatic recognition of idioms in scientific applications. In *Proceedings of the 17th international symposium on High performance distributed computing*, HPDC '08, pages 223–224, New York, NY, USA, 2008. ACM.

[23] K. Kennedy and J. R. Allen. *Optimizing compilers for modern architectures: A dependence-based approach*. Morgan Kaufmann Publishers Inc., 2002.

[24] O. Khalili, J. He, C. Olschanowsky, A. Snavely, and H. Casanova. Measuring the performance and reliability of production computational grids. In *Proceedings of the 7th IEEE/ACM International Conference on Grid Computing*, GRID '06, pages 293–300, Washington, DC, USA, 2006. IEEE Computer Society.

[25] B. Lu and J. Mellor-Crummey. Compiler optimization of implicit reductions for distributed memory multiprocessors. *Proceedings of the 12th International Parallel Processing Symposium (IPPS)*, 1998.

[26] G. Marin and J. Mellor-Crummey. Cross-architecture performance predictions for scientific applications using parameterized models. In *SIGMETRICS '04/Performance '04: Proceedings of the joint international conference on Measurement and modeling of computer systems*, pages 2–13, New York, NY, USA, 2004. ACM.

[27] M. Meswani, P. Cicotti, J. He, and A. Snavely. Predicting Disk I/O Time of HPC Applications on Flash Drives. In *Workshop on Application of Communication Theory to Emerging Memory Technologies with Globecom'10*, 2010.

[28] S. Muchnick. *Advanced Compiler Design and Implementation*. Academic Press, 1997.

[29] J. Odom, J. K. Hollingsworth, L. DeRose, K. Ekanadham, and S. Sbaraglia. Using dynamic tracing sampling to measure long running programs. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 59, Washington, DC, USA, 2005. IEEE Computer Society.

[30] B. Pottenger and R. Eigenmann. Idiom recognition in the Polaris parallelizing compiler. In *International Conference on Supercomputing*, 1995.

[31] A. Rifkin and B. L. Massingill. Performance analysis for archetypes. In *Proceedings of the 1998 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'98)*, 1998.

[32] P. Sassone and D. Wills. On the extraction and analysis of prevalent dataflow patterns. *Workload Characterization, 2004. WWC-7. 2004 IEEE International Workshop on*, pages 11–18, 2004.

[33] V. Taylor, X. Wu, J. Geisler, and R. Stevens. Using kernel couplings to predict parallel application performance. In *HPDC '02: Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing*, page 125, Washington, DC, USA, 2002. IEEE Computer Society.

[34] V. Taylor, X. Wu, and R. Stevens. Prophesy: an infrastructure for performance analysis and modeling of parallel and grid applications. *SIGMETRICS Perform. Eval. Rev.*, 30(4):13–18, 2003.

[35] J. Weinberg and A. Snavely. User-guided symbiotic spacesharing of real workloads. In *ICS '06: Proceedings of the 20th annual international conference on Supercomputing*, pages 345–352, New York, NY, USA, 2006. ACM.