Pir: Pmac's idiom recognizer

Conference Paper · September 2010

DOI: 10.1109/ICPPW.2010.36 · Source: DBLP

4 authors, including:

Catherine Rose Mills Olschanowsky
Boise State University
42 PUBLICATIONS **467** CITATIONS
SEE PROFILE

Laura Carrington
San Diego Supercomputer Center
76 PUBLICATIONS **1,948** CITATIONS
SEE PROFILE

PIR: PMaC's Idiom Recognizer

Catherine Olschanowsky, Allan Snavely
Department of Computer Science and Engineering
University of California, San Diego
La Jolla, CA
{cmills,allans}@cs.ucsd.edu

Mitesh R. Meswani, Laura Carrington
San Diego Supercomputer Center
University of California, San Diego
La Jolla, CA
{mitesh,lcarring}@sdsc.edu

***Abstract*—The speed of the memory subsystem often constrains the performance of large-scale parallel applications. Experts tune such applications to use hierarchical memory subsystems efficiently. Hardware accelerators, such as GPUs, can potentially improve memory performance beyond the capabilities of traditional hierarchical systems. However, the addition of such specialized hardware complicates code porting and tuning. During porting and tuning expert application engineers manually browse source code and identify memory access patterns that are candidates for optimization and tuning. HPC applications typically contain thousands to hundreds of thousands of lines of code, creating a labor-intensive challenge for the expert. PIR, PMaC's Static Idiom Recognizer, automates the pattern recognition process.**

PIR recognizes specified patterns and tags the source code where they appear using static analysis. This paper describes the PIR implementation and defines a subset of idioms commonly found in HPC applications. We examine the effectiveness of the tool, demonstrating 95% identification accuracy and present the results of using PIR on two HPC applications.

Keywords-automation; performance; static analysis; tuning

I. INTRODUCTION

Performance tuning of High Performance Computing (HPC) applications often centers on memory performance. Expert HPC application engineers manipulate parallel application code to best utilize complex hierarchical memory subsystems. Even with expert tuning, HPC application performance is often limited by memory performance. Recent fundamental changes to HPC resource architectures will potentially relieve this performance bottleneck, but at the same time complicate application porting and tuning.

In an effort to improve memory performance many new HPC resources include multi-core chips and specialized acceleration hardware such as Graphics Processing Units (GPUs) and Field Programmable Gate Arrays (FPGAs). Not all application code is suitable for execution on accelerator hardware; effective utilization of this hardware requires time consuming analysis of existing application code to determine sections whose performance will benefit from porting.

Early examples of these new architectures include Roadrunner at Los Alamos National Laboratory, which is composed of a dual core AMD Opteron processor paired with an IBM cell accelerator [1]. Roadrunner spent over a year in the number one spot on the Top500 list. Another example, the Convey HC-1 [2] employs hybrid-core computing, a multi-core processor paired with FPGAs. This machine has significant potential for large performance gains, according to performance models. However, in both of these cases achieving this performance requires that specific sections of code be chosen and ported for execution on the accelerator hardware.

The programmer must locate and customize application code to take advantage of the specialized hardware. She may replace common operations with highly optimized architecture-aware libraries such as BLAS or use special purpose hardware. Regardless of the optimization choice, a programmer must manually browse source code to identify opportunities.

This identification process is extremely labor-intensive. HPC applications consist of thousands to hundreds of thousands of lines of code that have been written to maximize performance on existing memory hierarchies; readability may be an afterthought.

Memory performance has long been a major obstacle to performance for HPC applications. In fact, the memory wall [3] has driven recent innovations such as hybrid-core computing. These innovations attempt to provide memory performance where traditional cache hierarchies cannot by allowing more flexibility and explicit control of data management and movement. Hence, memory access patterns are used to identify sections of code for optimization.

PIR (PMaC's Idiom Recognizer) is a static analysis tool that automates the process of identifying candidate sections of code. PIR automatically recognizes user-specified memory access patterns, called idioms, within application source code. This greatly reduces the amount of code that an expert must analyze manually.

PIR's design provides the flexibility to identify optimization opportunities for many different hardware configurations. The user provides descriptions of the idioms to be identified. As a starting point, PIR provides a set of commonly useful idioms and access to an Idiom definition syntax that allows for user customization of the idioms.

This paper presents the implementation of PIR, experimental evidence of its effectiveness and an example application tuning workflow using real HPC applications. We show that the idiom recognition is accurate within 95%. Furthermore, we demonstrate that PIR can be used to quickly identify areas of code with optimization potential.

The rest of this paper is organized as follows: Section II describes the implementation of the PIR tool and Section III describes the definition of a subset of commonly occurring idioms in HPC. Section IV describes our experiments to evaluate effectiveness of PIR and section V outlines a possible usage scenario. Section VI provides a summary of related work and Section VII describes conclusions and future work.

II. PIR APPROACH

PIR is a general framework that identifies common memory access patterns, idioms, within application code. The idioms of interest may vary between different architectures. For instance, one device may be specially equipped to handle random memory accesses while another is focused on speeding up consecutive accesses to large areas of memory. PIR, when supplied the appropriate idiom definitions, is capable of locating idioms for both.

PIR identifies patterns in application code by examining the compiler intermediate representation. An intermediate representation (IR) [4] is a data structure used internally by compilers after the code is parsed. An IR is a tree that contains control flow and data dependency information. This information is used by the compiler to optimize the source code. The optimizations are executed as transformations on the IR. Ultimately, the transformed IR is translated into machine specific assembly code. By focusing on the IR, PIR identifies patterns and structures suitable for porting before they are altered for machine-specific ISA mapping.

PIR is implemented within the GNU compiler collection (GCC) [7] taking advantage of its language and machine independent design. GCC is implemented in a modular fashion allowing the middle-end (the main body of GCC source code) of the compiler to be used for many languages. Figure 1 shows a high-level diagram of the GCC architecture. The front end contains language specific parsing code, the middle end is language and architecture independent and the back end contains architecture specific code. PIR is implemented as an optimization pass in the middle end of the compiler.

Although the operation of PIR is through a compiler, the result is not an executable, but a list of potential idiom locations. The implementation of PIR within GCC allows it to search for idioms within both FORTRAN and C without separate specialized code.

A series of optimization passes are performed by GCC after the GIMPLE form has been generated and transformed into Single Static Assignment (SSA) form. PIR is inserted directly after the loop optimization pass has been initialized. This allows access to all of the information necessary for loop level optimizations, but simplifies idiom recognition because the optimizations have not yet been executed.

A. GCC IR Illustration

The GCC IR is best explained via example; refer to the code snippet shown in Figure 2. This is a simple example of the stream idiom. More specifically it is a stream reduction, because the values are accumulated to a single variable rather than assigned to another array. This code is used throughout this section to illustrate the PIR approach.

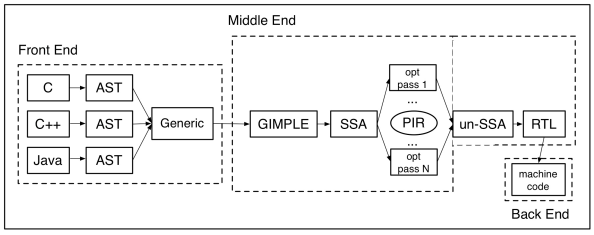

Figure 1. GCC Architecture Overview [5]

```
For i= 0,…,n
    sum = sum + A[i];
```

Figure 2. Example instance of the stream idiom

Understanding an idiom definition requires knowing what the IR for that idiom looks like. Figure 3 shows the GCC IR for this code snippet (the information for this diagram is printed by PIR). A node in the tree represents each operation and variable in the statement. For operations the child relationship indicates an operand. For variables, the child relationship indicates a step in the use-def chain.

The GCC IR uses a specific vocabulary, GIMPLE, to describe each node in the tree. PIR specifically examines portions of the tree that have a statement at the root (node 1). A statement implies that the result of an expression is being stored to a register or to memory. Other node names used in GIMPLE that are of interest include the following: ssa name (variable), binary_op, array, and const.

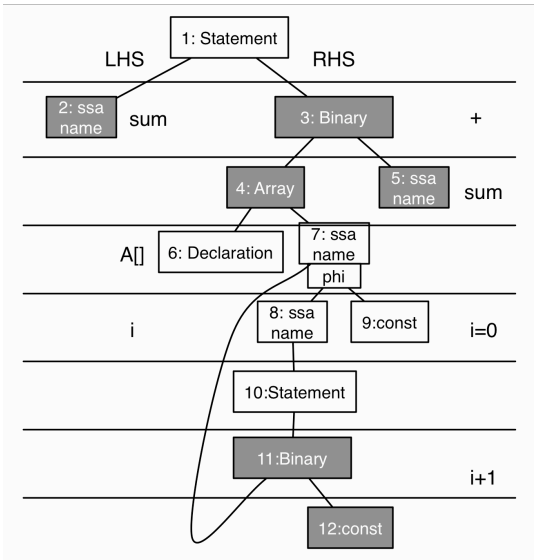

Figure 3. Example of the GCC IR

A statement node has two children, the first is the description of the left hand side, which can be as simple as an ssa name or it can be an expression describing a memory address, in this case it is the former. The second child is the right hand side. Node 2 represents the variable that will contain the result of the binary operation represented by node 3.

The right hand side of the statement describes the addition as well as the array access and computation of the loop induction variable. The direct children of the binary operation node (node 3) describe the operands. Node 6 is a declaration that describes the starting point of the array. Node 7 describes the index.

Nodes 7-12 represent the increment of the loop induction variable i, i.e., i= i+1. Node 7 is variable for the loop induction variable i. It is attached to a phi node because it has data dependencies going to two places. Node 9, const, is the original assignment of i to 0 at the beginning of the loop. The first child of the phi node (node 8) is the ssa name node that represents i. The first child of the binary operation node (node 11) is the same ssa name node.

Every statement in the application code has a tree similar to this created in the IR. They are organized into groups called basic blocks. Each basic block contains a series of statements that PIR examines.

B. PIR Implementation

PIR recursively traverses the IR for each statement searching for patterns. Each node of the GCC IR is well defined and has a small number of children. The GCC infrastructure provides the ability to walk the tree.

An important note about the implementation is that it looks for the existence of a pattern, not the absence of one. This means that the pattern needs to be carefully designed. For instance, looking for an ssa name on the left hand side of our working example, would match almost every statement in the application. By specifying that it must be found at level 0 the desired affect is achieved.

A few of the nodes require special handling, specifically ssa name. This node represents a variable and does not have any children. We follow the use-def chains in order to get to the statement that resulted in that variable. During this process it is important to be sure that we do not loop back on the same variable, as this would cause an infinite loop.

Labeling variables is necessary when recognizing a reduction pattern such as the one in the working example (Figure 3). It is necessary to know that the variable node represented on the left hand side is the same as the variable node that is the first operand to the binary operation on the right hand side.

It is possible to check for a dependence on the loop induction variable of a loop. The existing loop optimization code of GCC contains the dependency information. PIR accesses the internal data structure and internally names the variables in order to perform dependency checks and matches between the left and right hand side. This is necessary for identifying idioms such as transpose, described in the next section.

C. Idiom Definition Syntax

PIR allows users to create custom idioms using the idiom definition syntax. An idiom definition describes a pattern that, if present, indicates the presence of a specific idiom. Not every node in the tree has to be taken as part of the idiom signature. For the code snippet shown in Figure 3, only the grey nodes are taken as part of the signature. The definition must be specific enough to avoid false positives, and at the same time, avoid being overly complicated.

The idiom patterns are identified in a text file and read by PIR at runtime. This enables the patterns to be changed without requiring a recompile of the PIR source code.

Figure 4 shows one possible idiom definition for the running example, a stream reduction. The first entry specifies the label that each identified statement should be labeled with. The next group of statements describes the right hand side of a qualifying statement. The first field in the group states that there are four nodes specified on the right hand side.

GIMPLE specifies a class and a code for each node in the tree. The class is a grouping mechanism and the code is a more specific indicator. For example, the addition operator belongs to the class tcc_binary and has the code plus_expr. The subtraction operator also belongs to the tcc_binary class, but has the code minus_expr.

```
1:    label=STREAM_RED2
2:    rhs_count=4
3:    rhs_class[0]=tcc_binary
4:    rhs_code[0]=-1
5:    rhs_nesting[0]=-1
6:    rhs_iv[0]=-1
7:    rhs_class[1]=tcc_exceptional
8:    rhs_code[1]=SSA_NAME,0
9:    rhs_nesting[1]=0
10:   rhs_iv[1]=-1
11:   rhs_class[2]=tcc_reference
12:   rhs_code[2]=ARRAY_REF
13:   rhs_nesting[2]=0
14:   rhs_iv[2]=-1
15:   rhs_class[3]=tcc_binary
15:   rhs_code[3]=-1
17:   rhs_nesting[3]=-1
18:   rhs_iv[3]=0
19:   lhs_count=1
20:   lhs_class[0]=tcc_exceptional
21:   lhs_code[0]=SSA_NAME,0
22:   lhs_nesting[0]=-1
23:   lhs_iv[0]=-1
24:   idiom_end
```

Figure 4. One possible idiom definition for a stream reduction

Line 3 in Figure 4 indicates that a binary operation must take place on the right hand side of the candidate statement. The next line indicates that any binary operation is acceptable. Line 5 indicates the nesting requirement. The binary operation is the top most operation in this statement and so there is no nesting requirement here, indicated with a -1.

A loop induction variable is commonly referred to as the loop index. In the case of our example this is i. Line 6 indicates that there is no requirement that this binary operation depend directly on a loop induction variable.

The same pattern is continued for each of the nodes in the definition. A few interesting attributes are worth pointing out. On Line 8 the code (SSA_NAME) is followed by a comma and a 0. This indicates that the variable should be labeled. The labels are numeric and always start at 0. Line 21 uses the same pattern and the same label. This indicates that the variable on the left hand side and the right had side of the statement must match. Line 9 indicates that the ssa name node must be nested (be an operand of) the binary operation. In this case the 0 corresponds to the index used in the labels describing the binary node.

III. COMMON IDIOMS

PIR includes seven idiom definitions we have found to common in HPC applications. The idioms are described in the following section. All of the code samples are assumed to be part of a loop, i (and j) are loop induction variables.

- Stream: A[i] = A[i] + B[i]

The stream idiom includes accesses that step through arrays. In the above example two arrays are being stepped through simultaneously, but the stream idiom is not limited to this case. Stepping through any array in a loop where the index is determined by a loop induction variable is considered a stream.

- Transpose: A[i][j] = B[j][i]

The transpose idiom involves a matrix transpose, essentially reordering an array using the loop induction variable.

- Gather: A[i] = B[C[i]]

The gather idiom includes gathering data from a potentially random access area in memory to a sequential array. In this example the random accesses are created using an index array, C.

- Scatter: A[B[i]] = C[i]

The scatter idiom is essentially the opposite of gather. Values are read from a sequential area of memory and saved to an area accessed in a potentially random manner.

- Reduction: s = s + A[i]

A reduction can be formed from a stream, as in the working example, or a gather. It implies that the value returned from the read portion of the idiom is assigned to a temporary variable.

- Stencil: A[i] = A[i-1] + A[i+1]

A stencil idiom involves accessing an array in a sequential manner, including a dependency between iterations of the loop.

IV. EFFECTIVENESS AND PERFORMANCE

PIR facilitates application code optimization by listing specific areas of the code that fit a predefined pattern. The listing is provided in order to save the developer time when searching for optimization opportunities. Its effectiveness in this process depends on three abilities: finding all of the instances of a pattern (coverage), finding only the occurrences of the pattern (accuracy) and finding the patterns in a reasonable amount of time (performance).

Coverage and accuracy are evaluated by comparing the PIR generated idiom listing with a manually created listing. An expert read each line of code in a set of HPC benchmarks and generated a list of idioms for a set of patterns. False positives and misses were recorded during a comparison of the expert created and PIR generated listings.

The NAS Parallel Benchmarks (NPBs) [7] were used as the benchmarks. The NPBs represent some of the calculations that dominate execution time in HPC applications. A subset of the NPBs was used: BT, CG, EP, FT, MG, LU, and SP.

The evaluation is performed for the stream, gather and scatter idioms. These idioms provide a good evaluation of the tool because they are common in the NPBs, they require loop induction variable analysis, and they highlight the differences between FORTRAN and C arrays that complicate Idiom definition creation.

The included Idiom definitions are designed to find idioms within data that is statically allocated on the stack. This is an important distinction, because GCC represents data from the stack and the heap differently. It is possible to define Idiom definitions to recognize idioms on heap data, but the NPBs almost exclusively use the stack, therefore, only the stack data is included in this evaluation.

Table 1 summarizes the results of these experiments. The benchmarks are named in the first columns followed by their respective basic block and source code line counts. For each of the three idioms, stream, gather and scatter the number of lines marked as an idiom is shown in the column labeled PIR. This is followed by the reduction in the size of the search space resulting in using PIR as well as the number of false positives (FP) and the number of idioms missed by PIR (labeled miss).

Reducing the search space is a primary goal and PIR and the results show that this is done well. On average the search space is reduced by 95%. In some cases the entire benchmark is eliminated. The lowest reductions take place in stream. This is expected because stream is such a common pattern. The average reduction over gather and scatter is nearly 99%.

TABLE I. PIR ACCURACY AND COVERAGE (FP=FALSE POSITIVE,%RED.=SEARCH SPACE REDUCTION)

Benchmark/Idiom	Count		Stream				Gather				Scatter			
	BB	*Line*	*PIR*	*% Red.*	*FP*	*Miss*	*PIR*	*% Red.*	*FP*	*Miss*	*PIR*	*% Red.*	*FP*	*Miss*
BT	3958	9040	1440	84.1%	28	13	0	100.0%	0	0	398	95.6%	384	0
CG	1052	1787	108	94.0%	0	5	4	99.8%	0	1	6	99.7%	0	0
EP	342	317	0	100.0%	0	0	0	100.0%	0	0	0	100.0%	0	0
FT	1980	1993	58	97.1%	1	1	0	100.0%	0	0	0	100.0%	0	0
LU	3686	5926	1082	81.7%	2	0	0	100.0%	0	0	0	100.0%	0	0
MG	3416	2479	307	87.6%	4	2	15	99.4%	15	0	0	100.0%	0	0
SP	5388	4796	1814	62.2%	65	37	339	92.9%	309	0	317	93.4%	312	0

The data shows that coverage, the most important metric, is very good; PIR consistently finds over 95% of all the desired idioms.

The accuracy, expressed as false positives, is high for the stream idiom. The gather and scatter idioms highlight a specific problem with this approach for FORTRAN codes. Distinguishing an N-dimensional array access (N > 2) in FORTRAN from a gather is challenging. The compiler IR for each of these types of accesses is similar. The gather false positive rate is high because the gather signature also matches all N-dimensional array access.

It is possible to prevent the N-dimensional array false positives by narrowing the definition of gather. Doing this would cause some legitimate gather idioms to be missed as well, but depending on the user's goals, it is a potential solution.

Table 2 presents the time overhead for running PIR and looking for stream, gather and scatter. The overhead is on the scale of minutes rather than the hours or days it would take to achieve this manually. Compared to the time saved, the overhead is insignificant. The largest penalty, which took place compiling over 9000 lines of code, is approximately one minute.

TABLE II. PIR OVERHEAD

Benchmark/App.					
	Count		**Time (seconds)**		
	BB	***Line***	***GCC***	***PIR***	***Diff***
BT	3958	9040	0.17	65.60	65.43
CG	1052	1787	0.01	1.20	1.19
EP	342	317	0.01	0.36	0.35
FT	1980	1993	0.01	1.83	1.82
LU	3686	5926	10.52	26.83	16.31
MG	3416	2479	3.91	9.44	5.53
SP	5388	4796	13.52	108.16	94.64

PIR is an effective tool for narrowing the search space for optimization opportunities. Even with the high false positive rate for gather and scatter, the number of lines that need to be inspected is reduced by almost 99% in both of those cases.

V. USE CASE

PIR alone reduced the number of lines to be examined manually by 99% for gather and scatter. Used in conjunction with a tracing tool such as pmacInst [8] the optimization search space is reduced even further. This search space reduction is achieved by ranking the idioms found by PIR according to their execution frequencies, supplied by pmacInst. This process quickly identifies areas for optimization or eliminates the possibility that a specific application will be improved by porting it to use accelerator hardware.

We demonstrate this process and present the results of searching for gather and scatter idioms on two full scale HPC applications, FLASH and HMMER.

* FLASH: The FLASH [9] application is developed by the DOE-supported ASC / Alliance Center for Astrophysical Thermonuclear Flashes at the University of Chicago. This application is written using C and Fortran90 and has approximately 130,000 lines of code. We analyze FLASH version 3.1 in this paper.

* HMMER: The HMMER [10] application generates profile hidden markov models which are then used for searching sequence databases for homologs of protein sequences, and for making protein sequence alignments. This application is written in C language and consists of approximately 27,000 lines of code. We analyze version 2.3.2 MPI implementation [11] in this paper.

Gather and scatter are two fundamental operations that are popular in many parallel algorithms such as sorting and hashing [12]. These operations can potentially be optimized by special purpose hardware such as GPUs [13] or custom accelerators such as FPGAs.

```
Get()
{
for(i=0; i<n; i++)
  A[i]=B[C[i]];/*PIR ft.c:Get:440
GATHER*/
```

Figure 5. Sample Code Annotation

The gather and scatter idioms are identified in the application source code using PIR. The application is also analyzed using pmacInst. PmacInst rewrites application binaries and inserts instrumentation code for tracing. The traces contain execution frequency information that, when combined with idiom locations, can be used to pinpoint the most advantageous areas of application code to port to accelerator hardware.

Figure 5 depicts the use of PIR and pmacInst for analysis. Using the PIR tool, we automatically identify the occurrences of gather/scatter idioms. Table 3 shows a sample output from PIR, which identifies the file name, line number, function, and type of idiom identified. Using this we developed post-processing scripts to annotate the source code with the idiom information; an example code snippet is shown in Figure 6.

Next, in order to estimate the frequency of execution of each basic block we use the pmacInst tool to instrument every basic block of each application. Additionally, this tool was also configured to output the line number, function name, and file name of each basic block. Post-processing scripts then process the output of pmacInst and PIR. They generate a report about the basic block execution frequency of every function that contains idioms. Table 4 shows a sample output produced by this analysis.

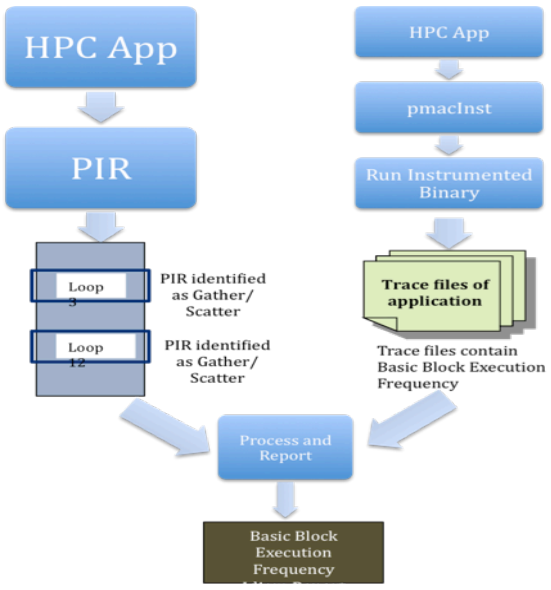

Figure 5. PIR and pmacInst Workflow

TABLE III. SAMPLE PIR OUTPUT

File Name	*Line #*	*Function*	*Idiom*
f1.c	2000	Add	scatter
f2.c	4400	Get	gather

TABLE IV. BASIC BLOCK FREQUENCY OUTPUT

File Name	Line #	Function	Number of Idioms	Basic Block Frequency %
f1.c	2000	Add	14	4.25%
f2.c	4400	Get	25	1.23%

The instrumented binaries were run on the NAVY system, Babbage [14] that has 3072 IBM POWER5+ processors. FLASH was run on 128 CPUs using the white dwarf input and HMMER was run on 4 CPUs. Execution of the binaries produces trace files that are then analyzed by post processing scripts to produce reports similar to the one shown in Table 4.

The search space for FLASH was reduced from over 125 thousand lines of code to 53. PIR identified a total of 354 possible instances of gather within FLASH; a 99% reduction in the possible search space for the user. Incorporating the data from pmacInst identified 53 instances that took place in code representing at least 1% of the total application basic block frequency. Of those 53 instances 36 were false positives leaving 17 statements to be examined.

The search results for scatter are similar to those for gather. PIR identified 221 instances of scatter, which pmacInst narrowed to 26 significant instances with 13 false positives.

In total the use of PIR and pmacInst reduced the search space for gather and scatter in FLASH down to 79 lines of code to be manually analyzed.

The Gather idioms located in flash were found in basic blocks comprising 8.6% of the basic block frequency and the scatter comprised 16.2%. Depending on the speed up possible by porting these statements to an FPGA may increase performance, however, the results for HMMER argue that porting is not advantageous.

No instances of the gather idiom were found in HMMER and only 4 instances of scatter were identified. Those 4 instances took place in code that comprised less than 1% of the basic block frequency. The use of PIR in this case is a major win. If gather and scatter are the only idioms that will benefit from a specific device, it is not worth pursuing for this application. This decision is reached with minimal effort on the developer's part.

VI. RELATED WORK

Idiom recognition has been implemented in several other tools, but PIR's focus on memory access patterns in HPC applications distinguishes it from past work.

Rose [16] is an open source compiler framework designed to generate source to source translators and analyzers. Compass [17] uses the Rose infrastructure to define simple patterns to verify the correctness of C, C++, and FORTRAN programs. The patterns used by compass do not refer to memory access patterns, rather coding rules. The goal is not to find optimization opportunities, but to certify code as secure.

Kowahati, et al. [18] developed an idiom recognizer built on the Java Just-In-Time compiler. They define idioms that can be potentially accelerated by special hardware-assist instructions. Their research improved previous work done for Java Compilers by recognizing patterns that are not exact matches with the specified idiom patterns. Using their approach, they were able to detect 75% more idioms than previous work in Java Compatibility Kit (JCK) API [19]. Idioms in the context of this project refer to topological attributes of the control flow graph of a program rather than access patterns.

Pinter et al. [20] developed an idiom recognizer that can identify parallelization opportunities within sequential C, C++, and FORTRAN programs.

Pottenger et al. [21] developed an idiom recognizer that worked in the Polaris compiler. Their idiom recognizer searches for opportunities to eliminate induction variables and find instances to parallelize reductions in FORTRAN programs.

Hiroyuki [22] developed an idiom recognizer that searches IR representation of FORTRAN programs to detect large arrays in numerical applications. Using their recognizer they show opportunities to improve the efficiency of BLAS routines.

VII. CONCLUSIONS AND FUTURE WORK

Recent innovations in HPC systems (i.e. hybrid-multi-core) have aggravated the challenge of optimizing memory intensive applications. The complexity of such systems makes the task of porting applications to the acceleration hardware of these hybrid-multi-core systems extremely time consuming. PMaC's Idiom Recognizer greatly reduces the manual effort required to pinpoint idioms within a large-scale code that are amenable to running on these hybrid cores.

We showed that the search space of large-scale HPC applications was reduced by over 99% for locating gather and scatter idioms. When used in conjunction with pmacInst the reduction is even more impressive; for FLASH the reduction is from ~130,000 lines of code to less than 100. This type of reduction can result in significant time savings for developers.

A potentially greater savings could be reached with the addition of a confidence metric to each idiom match. This will assist in evaluation false positives and allow the programmer to ignore tenuous matches. Work is also in progress for a tool that will automatically generate the idiom definitions from sample code. Not all developers are familiar with compiler intermediate representations, and automating this step will increase usability significantly. These additions, along with including a larger set of idioms definitions are planned for the future.

ACKNOWLEDGMENT

This work was supported by the DoD and used elements at the Extreme Scale Systems Center, located at ORNL and funded by the DoD. The software used in this work was in part developed by the DOE-supported ASC / Alliance Center for Astrophysical Thermonuclear Flashes at the University of Chicago. This research used resources of the National Center for Computational Sciences at Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

REFERENCES

[1] "Roadrunner." http://www.lanl.gov/roadrunner/

[2] "Convey," http://www.conveycomputer.com/

[3] Wulf, A.McKee S. Hitting the MemoryWall: Implications of the Obvious.. Computer Architecture News, 1995.

[4] Aho, A., Sethi R., Ullman J., Compilers Principles, Techniques and Tools. Addison-Wesley Publishing Company, 1986.

[5] "GCC Internals," http://gcc.gnu.org/onlinedocs/gccint/

[6] "GNU C Compiler Internals/GNU C Compiler Architecture - Wikibooks, collection of open-content textbooks," http://en.wikibooks.org/wiki/GNU_C_Compiler_Internals/GNU_C_Compiler_Architecture

[7] Bailey, D.H., Barszcz, E., Barton, J.T., Browning, D.S., Carter, R.L., Dagum, L., Fatoohi, R.A., Frederickson, P.O., Lasinski, T.A., Schreiber, R.S., Simon, H.D., Venkatakrishnan, V., Weeratunga, S.K. 1991.The Nas Parallel Benchmarks. *International Journal of High Performance Computing Applications, 1991*

[8] Tikir, M., Laurenzano, M., Carrington, L., and Snavely, A. The PMaC Binary Instrumentation Library for PowerPC. In *Workshop on Bi-nary Instrumentation and Applications*, 2006.

[9] "ASC Center for Astrophysical Thermonuclear Flashes," http://flash.uchicago.edu/website/home/

[10] Durbin, R., Eddy, S., Krogh, A., and Mitchison, G., "Biological sequence analysis: Probabilistic Models of Proteins and Nucleic Acids," *In Cambridge University Press*, 1998.

[11] "mpiHmmer," http://www.mpihmmer.org/

[12] Owens, J. D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A. E., and Purcell, T. J. A Survey of general purpose compuation on graphics hardware. In *Computer Graphics Forum*, 26.

[13] He, B., Govindaraju, N., Luo, Q., and Smith B. Efficient Gather and Scatter Operations on Graphics Processors. In *the proceedings of SC07*, 2007.

[14] " Babbage," http://www.navo.hpc.mil/babbage_about.html

[15] Schordan, M. and Quinlan, D. A Source-To-Source Architecture for User-Defined Optimizations. In *Joint*

Modular Languages Conference held in conjunction with EuroPar'03, August 2003

[16] "ROSE compiler project," http://www.rosecompiler.org/

[17] Quinlan, D.J., et al.: Compass user manual (2008), http://www.rosecompiler.org/compass.pdf

[18] Kawahito, M., Komatsu, H., Moriyama, T., Inoue, H., and Nakatani, T. 2006. A new idiom recognition framework for exploiting hardware-assist instructions. In *Proceedings of the 12th international Conference on Architectural Support For Programming Languages and Operating Systems* (ASPLOS-XII), 2006.

[19] Java Compatibility Kit, https://jck.dev.java.net/

[20] Pinter, S. S. and Pinter, R. Y. 1994. Program optimization and parallelization using idioms. In *ACM Trans. Program. Lang. Syst,* May. 1994.

[21] Pottenger, B. and Eigenmann, R. 1995. Idiom recognition in the Polaris parallelizing compiler. In *Proceedings of the 9th international Conference on Supercomputing* (*ICS'95*), July 1995.

[22] Hiroyuki, S. 2001. Array form representation of idiom recognition system for numerical programs. In *Proceedings of the 2001 Conference on Apl: An Arrays Odyssey* (*APL'01*), 2001.