Combining static and dynamic analysis to guide PGO for HPC applications: a case study on real-world applications

Youenn Lebras
University of Versailles
Versailles, France
Email: youenn.lebras@uvsq.fr

Andres S. Charif-Rubial
PeXL
Versailles, France
Email: ascr@pexl.eu

William Jalby
University of Versailles
Versailles, France
Email: william.jalby@uvsq.fr

***Abstract*—Modern high performance processor architectures tackle performance issues by heavily relying on increased vector lengths and advanced memory hierarchies to deliver high performance. This stresses the importance of data access optimization and efficient usage of the underlying hardware. Developers usually trust compilers to automatically address these performance issues, but unfortunately, compilers deploy static performance models and heuristics which, sometimes, remain conservative or even fail in the worst case. Moreover, manual optimization of production HPC codes is not only impractical, but impossible when having to manage multiple architecture dependent transformations. One way to assist compilers is to use Profile Guided Optimization (PGO). It allows the use of feedback data from dynamic profiling using a representative training dataset, for a given target application, enabling the compiler to refine its optimization choices and enhance application performance. But, PGO does not always consider certain metrics and is rarely aggressive enough regarding metric data collection. This bounds the transformation space and limits the compiler's ability to perform further optimizations. An additional option is to provide compilers with user guided assistance in order to enlarge the transformation space (i.e. specialization) and enhance the quality of optimizations.**

In this paper, we introduce ASSIST, a semi-automatic source-to-source manipulation taking advantage of static and dynamic profiling data produced by performance analysis tools. We demonstrate on real industrial class applications that by combining both static and dynamic analyses and by deploying simple transformations, ASSIST generates similar (and in some cases higher) performance speedups than Intel PGO. Furthermore, combining ASSIST and PGO allows to go a step further, increasing the performance substantially.

***Index Terms*—automatic, source-to-source, optimization, HPC, FDO**

I. Introduction

Modern high performance processor architectures rely heavily on increased vector lengths and advanced memory hierarchies to deliver high performance. This stresses the importance of data access optimization and efficient usage of the underlying available vector capabilities. Contemporary compilers are first to address these performance issues. Unfortunately, they suffer from two major limitations: first, the large size of the transformations space; second, the lack of solid guidelines (cost model) regarding the application of a transformation which is almost entirely based on static analysis feedback, rendering the transformation unrealistic in some cases.

With their knowledge of the code structure and using performance analysis tools capable of characterizing code behavior, developers may be able to guide the compiler to identify adequate transformations and their related parameters. They can annotate the source code either through custom directives [9], [24], [25], comments [12], [26] or by using Domain Specific Language (DSLs) [5], [10], [15], [18], [6]. Directives are simple but less powerful when compared to DSLs that are capable of handling very advanced patterns with additional complexity. From the point of view of a regular application developer, directives provide the best compromise (expressiveness vs. complexity). However, the resulting source code may end up bloated by extensive optimization transformations (i.e. tiling), handling special cases, or even useless modifications sometimes. This can worsen when users need to manage multiple target architectures (e.g. x86, ARM, ...). Also, most of these source code edits are the responsibility of the developers and will inevitably impact productivity and increase the risk of inserting useless or detrimental annotations, or even worse: bugs.

A very promising approach to relieve developers from this tedious annotation task is to use Feedback Data Optimization (FDO), also known as Profile Guided Optimization (PGO). Feedback data can be defined as any kind of information that can be extracted from a code and used to characterize its performance. *pgo* [21] and/or *autofdo* [11], is embedded within most production compilers: Intel, GCC, and, more recently, LLVM). A typical FDO process encompasses three steps: 1) producing an instrumented binary using a special compiler flag or multiple flags; 2) executing the binary in order to obtain a profile (feedback data); 3) using the obtained feedback data during a second compilation process to produce a new version that is expected to be more efficient. However, in the current FDO implementations the amount of information gathered at run time is limited and the available transformations space is fairly small. Both these limitations have a strong negative impact on the efficiency of the applied transformations. In this paper, we demonstrate that by being

more aggressive on information gathering and by combining static and dynamic information to refine the quality of transformations, substantial performance gains can be obtained. This paper presents ASSIST, an Open Source user guided source-to-source manipulation tool that allows automatic code transformations based on static and dynamic feedback. The tool aims at providing assistance with respect to productivity and performance efficiency. Our main contributions are the following:

- A novel study of how and when well-known transformations allow a performance gain on real-world HPC applications using a novel FDO source-to-source approach.
- A novel semi-automatic and user controllable method where, at different steps of the process, users can guide transformations using their expertise by choosing between different optimizations based on profiler results; if users deem a certain proposed optimization unnecessary, they can tell the tool to discard it.
- An FDO tool that combines both dynamic and static analysis information to guide code optimization while most other existing tools only use dynamic feedback.
- A verification system using a static analysis that checks if the proposed transformations do not have a negative impact on performances.
- A more flexible alternative to the compiler's PGO / FDO modes. Compilers that allow PGO search through a limited space and only perform dynamic analysis. Our method explores a larger search space in terms of performance analysis and is much more efficient, but its time overhead can be larger due to the large amount of gathered performance metrics. Last but not least we will show that our approach can be combined to PGO.

This paper is structured as follows: Section II provides an overview of our approach. Section III describes the design and implementation of the tool (ASSIST). Section IV presents the transformations provided by our tool and how they are triggered. Section V presents a study of experimental results. Section VI covers related work. And finally, section VII covers conclusion and future work.

II. Background and goals

ASSIST is part of the MAQAO tool-set [3] which focuses on performance evaluation and optimization of binary applications. The tool-set features multiple modules aiming at primarily pinpointing performance issues and providing users with hints or optimization propositions. MAQAO works at binary level (ELF, PE,...), taking advantage of evaluating low level constructs later to be executed on the underlying hardware. It is able to link those constructs to source code using debug information when available. Unfortunately, binary analysis is not always self explanatory to programmers and requires very good knowledge of architectural details which, in fact, are not always necessary to perform source level optimizations. When performing optimizations by hand on real-life applications, programmers face three main concerns:

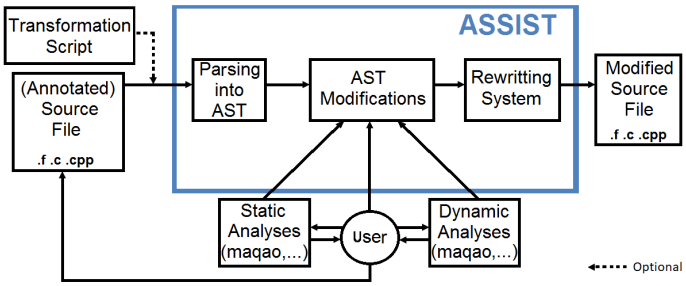

Fig. 1. Overview of the ASSIST process. The user decides what static and dynamic analyses have to be performed.

1) selecting which transformations to apply and in which order;
2) avoiding having to perform tedious or error-prone manual transformations.
3) minimizing code bloating due to transformations such as hand-coded (function/loop) specialization;

Our approach aims at enabling developers to improve their application's performance by providing a semi-automatic tool that helps in the selection of transformations, and then, under user control, implements them automatically.

III. Design and Implementation

In order to achieve the goals listed in the previous section, ASSIST must handle source code manipulation and harness the metrics and analyses produced by MAQAO tools. In this section, we present an overview of ASSIST and its infrastructure.

A. Overview

ASSIST is an open source semi-automatic FDO framework based on the ROSE [23] compiler infrastructure and integrated into the MAQAO tool-set. Figure 1 presents an overview of the main steps involved in the tool's operation. The following section provides more details on transformations and how to trigger them.

ASSIST provides expert programmers, as well beginners in optimization, a simple and flexible interface that offers three approaches to transform a source code. The first interface makes use of special directives (added by the user above a loop or a function) requesting from ASSIST to apply a specific code transformation. The second interface uses a script file instead of direct source file annotations. The third interface allows ASSIST transformations to be selected and triggered using profiling information gathered through static and dynamic performance analyses. Available analyses are based on CQA [4] (code quality analysis), VPROF [13] (value profiling) and DECAN [16] (binary modification). Users can select the metrics that will first be gathered by the MAQAO analyses modules and then automatically trigger ASSIST transformations. Feedback data can be coupled with user directives to guide transformations. During the modification process, the source code is parsed and transformed into an abstract

syntax tree (AST) that ASSIST will transform according to user directives and/or profiling results. As a semi-automatic tool, ASSIST can interact with the user during the AST modification and request information on potential issues with certain transformations (i.e. vectorization). We adopted the semi-automatic approach because it allows users to remain in control of the optimization process by evaluating the cost and the validity of the transformations. The tool provides cost estimates for the transformations, thus, users can assess and decide whether a transformation is worth applying or not. At the end of the process, the modified AST is parsed to generate a modified source file.

B. Compiler Infrastructure

Applying transformations to a given source code requires a set of front-ends. In our case, we give priority to HPC scientific applications, hence the focus on main HPC programming languages. What we would like to achieve with ASSIST using source-to-source transformations is independence from compiler-specific intermediate representations (IR) and directives.

Though there are many available source-to-source transformations APIs [17], [18], [8], [10], [20], we designed ASSIST around the AST infrastructure of the Rose Compiler Project. The Rose Compiler Project provides a rich Open Source (modified BSD license) API that suits our requirements.

IV. Transformations: What and How to Trigger Them

The main goal of ASSIST is to provide a better global strategy for selecting and triggering/applying standard code transformations. The search space for standard transformations is already large and a proper strategy is essential. Our approach will be to work on a per transformation basis determining what static and dynamic information need to be collected to assess whether the transformation is worthwhile. The assessment is made available to the user who will eventually decide to allow the tool to apply the transformation or not. This work does not propose new code transformations or new optimizations but presents customized standard optimizations, such as short vectorization, which aims at exploiting some key code characteristics: loops with very small trip counts.

A. Loop Count Transformation (LCT)

Knowing the loop iterations count enables to perform very efficient specialization. This can be exploited in many different ways through including compiler directives. Intel compilers offer the ability to specify a *loop count (min, max, avg)* directive. With such directives, the compiler will generate specific variants (with respect to unrolling, vectorization and prefetching) taking into account all the information the directive entails.

VPROF, the MAQAO value profiler is run on each loop and records, among other metrics, the minimum, maximum, and average number of iterations allowing ASSIST to automatically insert the corresponding directives. This transformation is considered the least aggressive and it allows to guide the compiler's decisions. It can therefore be used in addition to the usual optimizations of the compiler, PGO included. Section V shows the results obtained when combining the Intel PGO and the LCT where no case of performance degradation, due to optimization combination, was observed. On the contrary, both techniques tend to complete each other.

B. Tile & strip mine

The tile and the strip mine transformations can allow for substantial improvement in data locality. Using MAQAO, DECAN [16] generates several binary variants (DL1, REF, ...), then runs and compares them. For this use case, ASSIST uses the variant named DL1 in which all load and store memory operands are modified to target the same address across loop iterations. Thus, all data accesses (except for the first few iterations) will surely result in L1 hits. Comparing DL1 timings with original timings allows to measure precisely the impact of memory accesses, particularly in case the of "Perfect Blocking". The DL1 variant can be further refined by only targeting loads and stores on the same array in order to determine whether restructuring the array is worthwhile.

C. Specialization

Specialization is the process of creating particular versions of the same code by explicitly considering specific constant values for one or more variables. Specialization can be a very efficient transformation. Particularly, because it opens room for other transformations to be applied efficiently.

In the case of function specialization we will usually want to target specific value combinations. There are two main difficulties related to specialization: 1) detecting which variables are interesting to specialize and their values, and 2) minimizing the number of variables to specialize while keeping a generic code and having good performance results. Similarly to LCT, specialization can be triggered by using VPROF data. In addition to providing loop trip count, VPROF can profile specific variables to detect the most used values of these variables. To know which variable has to be profiled, we use ASSIST as a static analyzer. Either ASSIST focuses on functions and it will profile function parameters, or it focuses on loop nests and it will profile variables detected as interesting to specialize. It will be mostly variables governing conditionals and loop bounds. ASSIST can then insert calls to the VPROF library for the interesting variables. The resulting code will be compiled and executed to obtain desired values. Finally, specialization is triggered using results from the value profiling. Specialization will be essentially triggered on non uniform value distribution. A particular interest will be put on variables with "dominating" single values.

D. Short vectorization transformation (SVT)

The short vectorization transformation allows to improve vectorization of a loop with a very small trip count (typically less than 16). Such cases can be detected using MAQAO CQA which statically analyses loops and computes vectorization

```
double *a, *b;
...
#pragma MAQAO SHORTVEC=AVX2
for (int i=0 ; i < 7; i++ ) { a[i] += b[i]; }
```

(a) Before short vectorization

```
double *a, *b;
...
#pragma simd
#pragma vector unaligned
for (i = 0; i < 4; i++) { a[i] += b[i]; }

#pragma simd
#pragma vector unaligned
for (i = 4; i < 6; i++) { a[i] += b[i]; }
a[6] += b[6]
```

(b) After short vectorization

Fig. 2. Example of short vectorization on x86_64 performed by ASSIST

efficiency coupled with VPROF to obtain the trip count of that loop. To overcome this vectorization issue, ASSIST use the SVT which performs the following steps: force the compiler to vectorize the loop using the *SIMD* directive; prevent peeling code from being generated using the *vector unaligned* directive; and adapt the number of iterations to the vector length. Figure 2 illustrates this transformation.

V. Experiments

In this section we present results obtained using ASSIST and compare them with Intel compiler PGO mode (denoted IPGO). Intel compilers are neither open source nor free, but they provide the best performance in our tests (compared to GCC and LLVM). For IPGO, the use of profiling data enables some specific optimizations but can also modify the behavior of other optimizations: 1) using feedback data on function entry counts. Function grouping is done to put hot/cold functions adjacent to one another; 2) value profiling of indirect and virtual function calls is done to specialize indirect function calls for a common target; 3) The intermediate language is annotated with edge frequencies and block counts which are then used to guide a lot of the optimization decisions made by other passes of the compiler. Our goal is not to "mimic" IPGO, but rather to present a complementary approach which goes beyond the observed limitations. All the measurements presented below were gathered on an Intel(R) Skylake SP based machine (Intel Xeon Platinum 8170 CPU@2,10GHz) with Intel compiler version 17.0.4. Multiple executions (exactly 31) were performed to reach statistical stability and avoid noisy measurement data. Also, this section presents experimental results of transformations provided by ASSIST based on feedback data and user insight. This study is application-centric, we have looked for an approach to get a good performance gain at minimal cost; we start from what the application needs, based on what MAQAO profilers return, to trigger the right transformation for each application and thus limit the search space and avoid blindly testing different useless transformations.

	AVBP NASA	AVBP TPF	AVBP SIMPLE	Yales2 3D Cylinder	Yales2 1D COFFEE
Nb loops	149	173	158	162	122

TABLE I
Number of loops processed by ASSIST LCT for each application and test case.

A. Application pool

Four full industrial class applications were used to test our approach: YALES2 [7], a numerical simulator of turbulent reactive flows.ASSIST has been tested on two of their datasets named "3D_Cylinder", a pure CFD computation, and "1D_COFFEE", a combustion computation. The application is written in Fortran 2003 and contains approximately 276 000 lines of code. AVBP [2], a parallel CFD code developed by CERFACS that solves the three-dimensional compressible Navier Stokes equations on unstructured multi-element grids.ASSIST has been tested on three representative datasets namely: SIMPLE (helicopter chamber demonstrator combustion simulation), NASA (NACA blade simulation) and TPF (large flow simulation). The application is written in Fortran 95 and contains approximately 275 000 lines of code. ABINIT [1], a package allowing users to find the total energy charge density and electronic structure of systems made of electrons and nuclei.The application contains approximately 807 000 lines of Fortran 90 code. Convolutional Neural Network (CNN), the state-of-art Deep Neural Network for image recognition.The CNN code refers to the one used in [19] and the layers used are the GoogleNet_V1. The convolution technique consists of executing all CNN layers one after the others with different filter sizes (1x1, 3x3 and 5x5). This codelet is written in C and contains 450 lines of codes.

B. Impact of loop value profiling

Our first FDO optimization uses loop trip counts information obtained by value profiling using MAQAO VPROF. When loops exhibit a complex control flow due to multi-versioning, knowledge of the trip count can help the compiler simplify the decision tree. Figure 3 presents speedups obtained with LCT, IPGO and the combination of both for each application/dataset. For these applications, the combination of LCT and IPGO reach a speed up of 14% for a sequential YALES2 run with the 3D_Cylinder dataset. To ensure that the optimization is still efficient in parallel, these figures present the speedup of the LCT, IPGO and both.In most cases, the speedup decreases when the number of processes increase. This is due to the communications which proportionally increases (see MPI time plots) at the same time and take most of the execution time. On the contrary, for Yales2 with 1D COFFEE dataset, an increasing speedup according to number of processes can be observed. This is due to an Intel compiler optimization on an Intel library function that performs a copy of a string used for all communications. The higher the number of communications, the more often this function is called. By providing

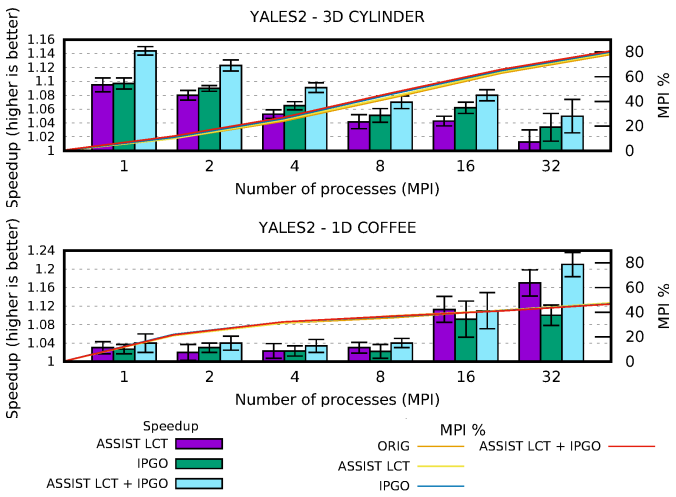

Fig. 3. **Histograms**: impact (speedup) of ASSIST LCT, IPGO and combination of both compared with original version for the same number of threads of two datasets Yales2 (Higher is better). **Error bars** represent original version / minimum speedup and original version / maximum speedup. **Plots**: Percentage of execution time spent in MPI.

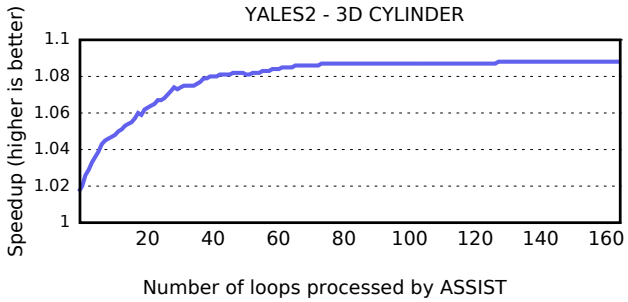

Fig. 4. Cumulated speedup versus number of loops processed by ASSIST, sorted by coverage, on Yales2 using the 3D Cylinder test case.

the compiler with iteration count of loops containing this function, the compiler can perform advanced optimizations. After applying ASSIST LCT, we used our verification system based on CQA to statically verify that the compiler did not generate a worse performing code. The verification system is not yet totally implemented, so we only apply it on hot loops and confirm that the transformation does not downgrade performances. The strength of this transformation comes from the number of loops processed by ASSIST; as shown on figure 4, the first twenty loops provide more than fifty percent of the total speedup gain but 130 loops are necessary to reach a maximum speedup.

This study shows that providing the compiler with loop trip count feedback (minimum, maximum and average values) results in significant performance gains. When compared with IPGO, performance gains are lower but we should be keep in mind that IPGO and ASSIST LCT are using different optimizations. The most important point is that both can be combined and their combination leads to higher gains.

C. Impact of specialization

The following examples show how specialization alone, or coupled with other transformations, can provide a significant performance gain.

*1) **AVBP**:* In this example, MAQAO indicates that in the ten most time-consuming functions there are loop nests with poor vectorization efficiency and a low trip count for innermost loops for the three datasets: NASA, TPF and SIMPLE. ASSIST has been used to couple both specialization and SVT on these functions. Loop and function specializations have been first applied separately. Then, the SVT has been performed on the most efficient version. Figure 5 only presents results on the dataset SIMPLE because it is the most relevant.

Figure 6 compares the speedup ratios of each version (LCT, IPGO, LCT + IPGO and SVT). For the TPF dataset the SVT allows to gain as much as the combination of LCT and IPGO. But for the the NASA dataset the best of LCT+IPGO only allows to reach almost half of the speedup obtained with SVT for one MPI thread. It is more blatant with the SIMPLE dataset, the speedup of LCT and IPGO does not reach more than 2% individually and 4% when combined, contrary to ASSIST SVT which reaches a 12% speedup for the SIMPLE dataset. When the compiler fails to vectorize a loop properly, SVT is very effective given that it explicitly exposes a simpler loop structure with no peel or tail loops to the compiler. There are two main reasons why the compiler does not vectorize: first, the dependence analysis reveals dependencies preventing vectorization, and second, the cost model used by the compiler gives estimates that vectorization is not beneficial. On other cases, compiler performed an outer vectorization on loops with a small number of iterations, CQA detects a bad "vectorization efficiency" on these loops. CQA offers multiple vectorization metrics such as vectorization-ratio or a vector-efficiency ratio on loads, stores, etc. allowing to assess the performance level obtained. In our case, we use these metrics to provide ASSIST with quality estimates of the vectorization carried out by the compiler to perform or not a good vectorization and finally to trigger the transformation. The short vectorization transformation forces the compiler to vectorize small loops with a small number of iterations; the compiler also fully unrolls these loops. After transformations, we use our verification system with CQA to validate the transformations. Indeed, before transformations, CQA detects only 33% of vectorization and after, CQA reports the loop as fully vectorized.

To apply SVT, loop bounds have to be known. To set these bounds, we specialize functions on one side, and loops on the other; and apply the SVT on the better specialization for each function. Figure 5 presents speedups obtained at each step to show their individual impact, we add ASSIST LCT and IPGO for comparison. We observe that SVT can rise to up to 2.6x speedup while the loop and function specializations only achieve, at best, a speedup of 1.5x. Performing only loop or function specialization may be counterproductive in some cases because of the induced complexity of the control flow, if

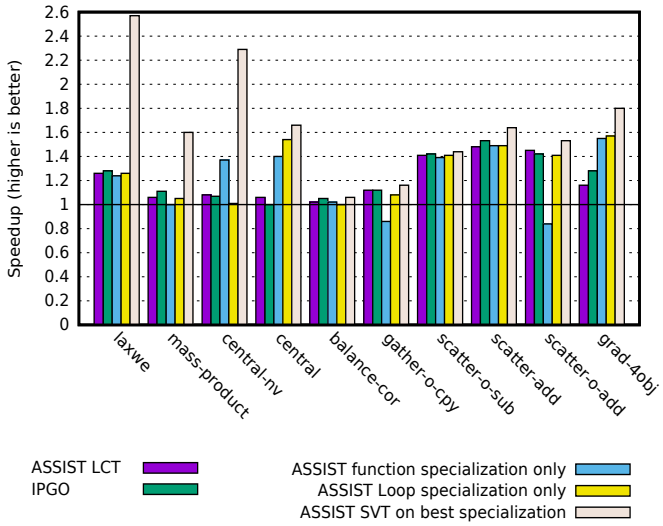

Fig. 5. Speedups by function before and after applying transformations with ASSIST (SVT, function/loop specialization, LCT) and IGO compared with the original version (higher is better) on AVBP using the SIMPLE test case (sequential version).

loop id	# ite. min	# ite. max	# ite. avg	Potential speedup	Coverage
16182	4	4	4	4.00	11.25
13752	4	4	4	2.67	3.46
13692	4	4	4	4.00	2.98
13902	3	3	3	6.67	2.51
13641	4	4	4	4.00	1.52
2587	5	5	5	2.00	0.97
2690	4	4	4	6.40	0.61
2551	5	5	5	8.00	0.37
2723	5	5	5	2.00	0.35
2308	3	3	3	8.00	0.33

TABLE II
CQA & VPROF METRICS OF LOOPS OF THE HOTSPOT FUNCTIONS OF AVBP, WITH THE SIMPLE DATASET, BEFORE APPLYING THE SVT.

no further induced optimizations are possible. Table II presents metrics from CQA and VPROF of loops before applying the SVT; it is these metrics that motivate our choice to use SVT. SVT only has been performed on AVBP because the other applications have different bottlenecks or because SVT would be counterproductive. For example, in case of Yales2, SVT has not been applied because CQA indicated that vectorization will lead to the use of scatter/gather instructions which are costly and make vectorization not beneficial.

*2) **ABINIT**:* In this example, ASSIST is used as a semi-automatic tool and is fully driven by the user. At first, a full profiling of the code is performed followed by value profiling on one of the main hotspots of the application. Three input parameters were found to be of importance.

First, the function can be called with two different types of input data, either real-valued data or complex-valued data. A given test case will almost exclusively use one or the other. As those data are expressed as an array with one or two elements in a part of the code, specialization of this value simplifies address computations and vector accesses by making the stride

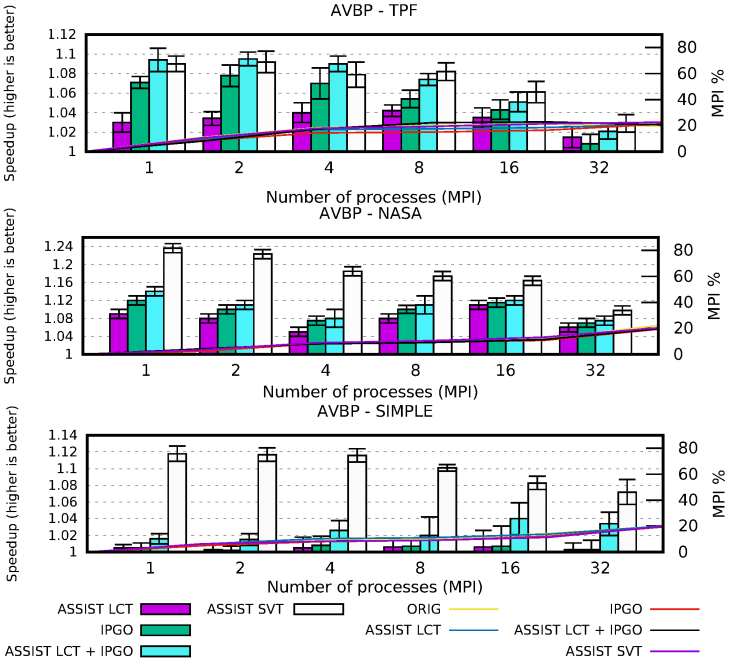

Fig. 6. **Histograms**: Speedups of ASSIST SVT & LCT, IPGO and ASSIST LCT + IPGO compared with the original version (higher is better) on three AVBP datasets. **Plots**: Percentage of execution time spent in MPI.

a compile-time constant rather than a dynamic value.

Second, multiple variants of the algorithm are implemented in the function. Which exact variant is used, depends on two integer parameters. Again, a given test case is usually heavily biased towards a small subset of possible cases. Specialization to one case removes multiple conditionals. As the loop nests for a given case appear in different branches, this removal of conditionals exposes the true dynamic chaining of loop nests to the compiler with no intervening control-flow break.

Once specialized with ASSIST, the function becomes much simpler to study. A study using MAQAO DECAN indicated that a large array is updated in its entirety inside a loop, a bad pattern for cache usage where tiling would be very beneficial. Loop tiling makes it possible to update the array by blocks, and to only scan and update the array once. While this work would not be particularly difficult to do by hand, more than two dozen variants of the loop nest with similar properties appear in the original function. As the transformed loop adds an extra loop to the nest, this complicates indexes and requires a remainder loop. It is much easier and much more reliable to automate the transformation process. Speed up results are shown in figure 7. We added IPGO to show the potential of our approach. Specialization offers a small gain but the dominant issue is still the time spent in the critical loop nest. Adding tiling offers a large gain of almost 1.8x in total by significantly reducing the memory bandwidth of the critical loop nest. Despite the complexity of the original function, ASSIST should make it easy to apply the same transformations to other possible uses of the function for other test cases of the ABINIT code.

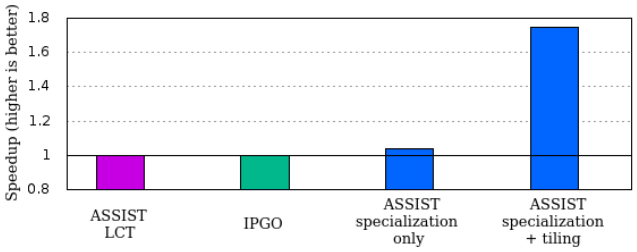

Fig. 7. ABINIT - Ti-256 - Speedups of IPGO, ASSIST LCT, specialized with ASSIST, specialized and tiled with ASSIST compared to the original version

*3) **CNN**:* In this example, our target loop nest is composed of seven nested loops and ASSIST is used in two steps: first, as an automatic tool, using the automatic specialization to detect variables that can be automatically specialized. In this case, ASSIST found that by specializing variables for certain values, it is possible to set bounds of the two innermost loops of the loop nest and also possible to remove the *if* statements that are in these two loops; then, as users, we know that two variables - which are computed inside the loop nest - only have three possible values for most layers. These are calculated within the loop nest which prevents the previous automatic specialization. After both specializations, the loop nest increased from 30 lines to 922 lines to handle all cases. This transformation can hardly be manually done without making mistakes.

Figure 8 presents speedups after the two specializations compared to the original version. Specializations offer a gain between 1.4x and 5.4x on all tested layers by creating multiple, less complex, versions of the loop nest that the compiler can more easily optimize. Layers used are those with a (1x1) and (3x3) filters. IPGO does not appear on this figure because it does not gain any performance.

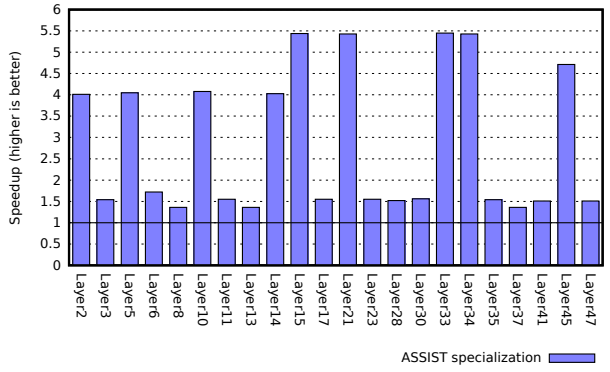

Fig. 8. Convolution Neural Network - Speedup of GoogleNet_V1 layers after specialization, compared to the original version.

VI. Related Work

The originality of the approach presented in this paper lies in the combination of both source-to-source transformations with user guidance and FDO approaches. There are several tools allowing to perform source-to-source transformations [12], [5], [10], [14], [9], [22]. These tools provide specific transformations (i.e. adding parallelism) or provide APIs to let users define their own transformations. Very few of these tools use static and dynamic information to perform transformations. Usually, they rely on empirical optimizations by creating multiple variants and executing each one before choosing the best version. Our approach opts for a cheaper and more straightforward path using FDO and takes advantage of static and dynamic analyses. That way we can assess the quality of the code generated by the compiler (using CQA) and get execution behavior metrics. The only tools performing transformations based on static and dynamic information are FDO tools such as PGO(e.g. Intel, GCC, LLVM) or AutoFDO [11] (available for GCC and LLVM). AutoFDO is the name of an in-house FDO deployment system proprietary to Google. Compared to PGO, AutoFDO exploits hardware counter profiles. In both cases feedback/profile data is injected early in the intermediate representation of the compiler so that all the optimization passes can take advantage of it. Our approach aims at helping modern compilers by not injecting data using a specific format, but rather by working at source level to avoid being compiler dependent. From a performance point of view, both approaches are complementary.

VII. Conclusion and Future Work

We have shown the efficiency of our approach: how and when already known transformations allow to gain speedup on real-world HPC applications by using static and dynamic feedback data or user guidance. Though no new optimization techniques are developed, new combinations of transformations are shown to be worthwhile. Moreover, given that our work is at source-level, our approach allows to remain portable across compilers and architectures.

The tool presented in this article provides the foundation for a more complete semi-automatic tool which will combine more of FDO and user knowledge. As future work we plan to harness all the available metrics and dynamic analysis existing in MAQAO including those using hardware counters to perform and automate more optimizations. We also plan to improve the specialization by searching where we can ”backtrack” in a loop nest to specialize. For example, if we detect that the innermost should be specialized, we may want to detect if we can move up the specialization in the loop nest without being hindered by an assignment which affects a variable to specialize.

Acknowledgements

This work was carried out within ECR (Exascale Computing Research). The authors thank INTEL, CEA and UVSQ for their financial support.

REFERENCES

[1] ABINIT. https://www.abinit.org/.

[2] AVBP. http://www.cerfacs.fr/avbp7x/.

[3] D. Barthou, A. Charif-Rubial, W. Jalby, S. Koliai, and C. Valensi. Performance tuning of x86 openmp codes with maqao. In *Parallel Tools Workshop*, pages 95–113, Desden, Germany, Sept. 2009.

[4] A. Charif-Rubial, E. Oseret, J. Noudohouenou, W. Jalby, and G. Lartigue. Cqa: A code quality analyzer tool at binary level. In *HiPC*, pages 1–10. IEEE Computer Society, 2014.

[5] C. Chen, J. Chame, and M. Hall. Chill: A framework for composing high-level loop transformations. 2008.

[6] J. R. Cordy. Source transformation, analysis and generation in txl. In *Symposium on Partial Evaluation and Semantics-based Program Manipulation*, PEPM '06, pages 1–11, New York, NY, USA, 2006. ACM.

[7] Coria. http://www.coria-cfd.fr/index.php/YALES2.

[8] Dave and al. Cetus: A source-to-source compiler infrastructure for multicores. In *Computer*, pages 36–42, 2009.

[9] R. Dolbeau, S. Bihan, and F. Bodin. Hmpp: A hybrid multi-core parallel programming environment. In *(GPGPU 2007*, volume 28.

[10] C. L. et V. Adve. Dms/spl reg: program transformations for practical scalable software evolution. In *ICSE'04.*, pages 625–634. IEEE.

[11] Google. https://github.com/google/autofdo.

[12] A. Hartono, B. Norris, and P. Sadayappan. Annotation-based empirical performance tuning using orio. In *ISPA 2009*, pages 1–11.

[13] S. Henry, H. Bollore, and E. Oseret. http://www.hsyl20.fr/home/files/papers/shenry_2015_vprof.pdf.

[14] Irigoin and al. Interprocedural analyses for programming environments. In *Workshop on Evironments and Tools For Parallel Scientifc Computing*, Saint-Hilaire du Touvier, France, 1992.

[15] P. Klint, T. van der Storm, and J. Vinju. Rascal a domain specific language for source code analysis ad manipulation. In *IEEE International Working Conference on SCAM 2009*, pages 168–177. IEEE.

[16] S. Koliaï, Z. Bendifallah, M. Tribalat, C. Valensi, J.-T. Acquaviva, and W. Jalby. Quantifying performance bottleneck cost through differential analysis. In *ICS '13*, pages 263–272. ACM.

[17] C. Lattner and V. Adve. Llvm a compilation framework for lifelong program analysis and transformation. In *CGO'04*. IEEE.

[18] R. v. M. Bravenboer, K. T. Kalleberg and E. Visser. Stratego/xt 0.17. a language and toolset for program transformation. In *Science of Computer Programming*. Elsevier, 2008.

[19] A. Mandal and al. Using dynamic compilation to achieve ninja performance for cnn training on many-core processors. In *Europar*. IEEE, Aug. 2018.

[20] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *CC'02*, pages 213–228, Univ. of California, Berkeley, USA. Springer.

[21] D. Novillo. Samplepgo: The power of profile guided optimizations without the usability burden. In *LLVM-HPC '14*, pages 22–28, Piscataway, NJ, USA. IEEE Press.

[22] Paraformance. http://paraformance.weebly.com/.

[23] Quinlan and al. Rose: Compiler support for object-oriented framework. In *Parallel Processing Letters*, pages 215–226, Lawrence Livermore National Laboratory, Livermore, CA, USA, 2000. World Scientific.

[24] R. Suda, H. Takizawa, and S. Hirasawa. Xevtgen: Fortran code transformer generator for high performance scientific codes. In *International Journal of Networking and Computing*, pages 263–289, 2016.

[25] X. Xiao, S. Hirasawa, H. Takizawa, and H. Kobayashi. An approach to customization of compiler directives for application-specific code transformations. In *International Symposium on Embedded Multicore/Manycore SoCs*, pages 99–106, Sept 2014.

[26] Q. Yi. Poet: A scripting language for applying parameterized source-to-source program transformations. In *Software Practice And Experience*, pages 675–706, Univ. of Texas at San Antonio, USA, 2012. John Wiley and Sons.