Chapter 1
ASSIST: A FDO source-to-source transformation tool for HPC applications

Youenn Lebras[1,3], Andres S. Charif Rubial[2,1], Romain Dolbeau[4], and William Jalby[1,3]

Abstract The complexity and the diversity of computer architectures have dramaticaly evolved over the last decade, which makes it impossible to manually optimize codes for all these architectures. In addition, compilers must remain conservative with respect to their optimization choices because of their static cost model. One way to guide them is to use feedback data from data profiling of a representative training dataset (FDO/PGO) for a given application. It then becomes possible, based on that knowledge, to add specific compiler directives and/or flags to enhance performance. Moreover, automatic transformations simplifying portions of the application (e.g. specialization) can be applied. In this paper we present ASSIST, a directive-oriented source-to-source manipulation tool that aims at providing such assistance. The tool is integrated into the MAQAO toolset and takes advantage of all the available static and dynamic profiling data produced by the other tools. It also features a set of code transformations triggered by directives. The combination of both leads to an autotuning process that helps users to keep their code as generic as possible whilst also benefiting from a performance gain related to feedback or user knowledge. We demonstrate how we can build a compiler's PGO-like tool and compare our first results to the Intel compiler PGO mode.

1.1 Introduction

The new generation of high performance cores/processors is heavily relying on increased vector length and advanced memory hierarchies to deliver higher performance levels. Such trends stresses the importance of data access optimizations and vectorization. The compiler is the first classical approach to address these issues. Unfortunately they suffer from two major limitations: first their search for code transformations is de facto limited because searching through a huge space of transformations without specific "tips" is

University of Versailles SQY · PeXL · Exascale Research Computing · Atos

extremely expensive and second because the choice of the applied transformation is entirely based on static information, running the risk of missing the right target.

The code developer with his knowledge of the code which might be augmented by the use of performance tools to characterize code behavior can help the compiler in identifying the right transformations. He can annotate source code either through custom directives [28,29,35], comments [16,36] or using Domain specific Language (DSLs) [7,8,11,21,26]. Directives are simple but less powerful when compared to DSLs which can handle very advanced patterns at the price of complexity. From the point of view of a a regular application developer, directives provide the best compromise (expressiveness v.s. complexity). However, the resulting source code may end up bloated by optimization transformations (e.g. tiling), special cases or even useless modifications. It is even worse if users need to target multiple architectures (e.g. x86 and GPU or ARM). Finally, all of these source code edits are put on developers shoulders, impacting its productivity, creating the risk of inserting useless or detrimental annotations or much worse, introducing bugs.

A very promising approach to relieve the user from these tedious edits is to use feedback data optimization (FDO). Feedback data is any kind of data that can be gathered on a code and can be used to characterize it from a performance perspective. It should be noted that in the literature FDO and Profile Guided Optimizations (PGO) have the same meaning. We will use FDO in the rest of this article because, in our opinion, it is more generic. For instance feedback data could be a small trace which differs from a profile. One example of FDO is the FDO modes embedded with production compilers (Intel, GCC and more recently LLVM) known as *pgo* [22] and *autofdo* [6,17]. A typical FDO process encompasses three steps: producing an instrumented binary using a special compiler flag(s); executing the resulting binary to obtain a profile; and finally, using feedback data during the compilation process to produce a new version that is supposed to be more efficient. However, in the current FDO implementations the level of information gathered at run time is limited and second the transformation space searched is also limited. Both limitations have a strong detrimental impact on the efficiency of the transformations applied. We will demonstrate that by being more aggressive on information gathering combining static and dynamic information, and on code transformations, substantial performance gains can be obtained.

This paper presents ASSIST, a directive-oriented source-to-source manipulation tool. It is able to guide code transformations based on static and dynamic feedback. It aims at providing assistance with respect to productivity and performance efficiency. The main contributions of our tool are to provide: a new open source FDO tool using both static and dynamic feedback while existing ones only use dynamic feedback; a more flexible alternative to compilers PGO/AutoFDO modes while being complementary; elaborated transformations such as loop and function specialization including our block vectorization transformation which helps the compiler to harness vectorization.

This paper is organized as follows: Section 1.2 provides an overview of our approach. Then section 1.3 describes the design and implementation of the tool. The following section 1.4 presents the available transformations. In section 1.5 we will study the experimental results. Related work is listed in section 1.6 before concluding and mentioning future work in section 1.7.

1.2 Background and goals

The MAQAO toolset [2] focuses on the performance evaluation and optimization of binary applications. The toolset features multiple tools [5, 19, 20, 30] which share the same rationale, namely pinpointing issues at source level and providing users with hints and even workarounds to be applied. In order to efficiently use these tools, a methodology [3] has been proposed. It aims at providing a way to filter all the data collected from the performance evaluation tools and classify them according to their return on investment (ROI) metrics.

Working at binary level has the advantage of evaluating the code that will really be executed (i.e. after compiler modifications). However, the main drawback is that we do not have access to the source code. A match between assembly level and high level source structures like functions/loops has to be based on debug information provided by the compiler. According to the optimization level, debug information is more or less accurate. This is due to the transformations/optimizations (e.g. inlining) performed by the compiler. It is also impossible to control all code properties that could help to provide more accurate results when combined with binary analyses. Enabling MAQAO to deal with source code would allow more accurate analyses. MAQAO can pinpoint different kinds of performance issues (i.e. diagnosis). The next step is to try to fix them at source level.
When performing optimizations on real applications we face three main concerns: selecting which transformations to apply to fix issues; minimizing code bloating due to transformations like hand-coded (function/loop) specialization; avoiding having to apply tedious (when not error-prone) transformations. The main goal of our approach is to help users increase performance without reducing the programming productivity.

1.3 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. Then, we will explain the choice of the selected compiler structure. Finally we will show how ASSIST can benefit from its integration into MAQAO and vice versa.

1.3.1 Overview

ASSIST is an open source FDO tool and framework based on the Rose [27] compiler infrastructure and integrated into the MAQAO [2,32] toolset. More details are provided in the next subsections.

Figure 1.1 presents an overview of the steps involved in the tool's operation. The following section will provide examples illustrating this process.

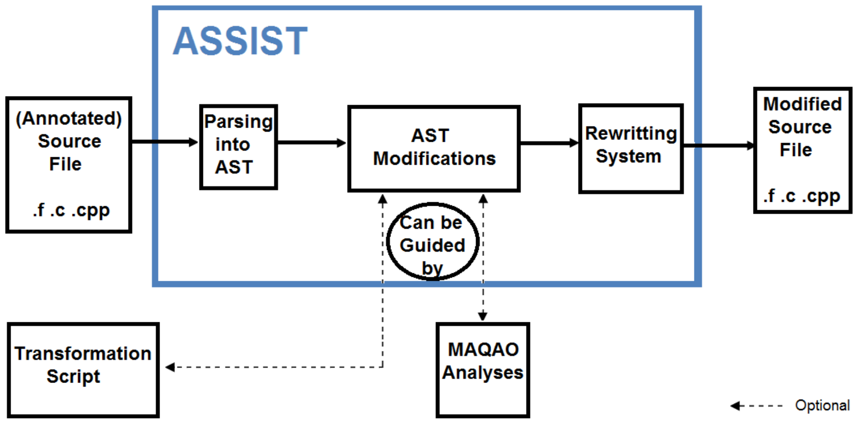


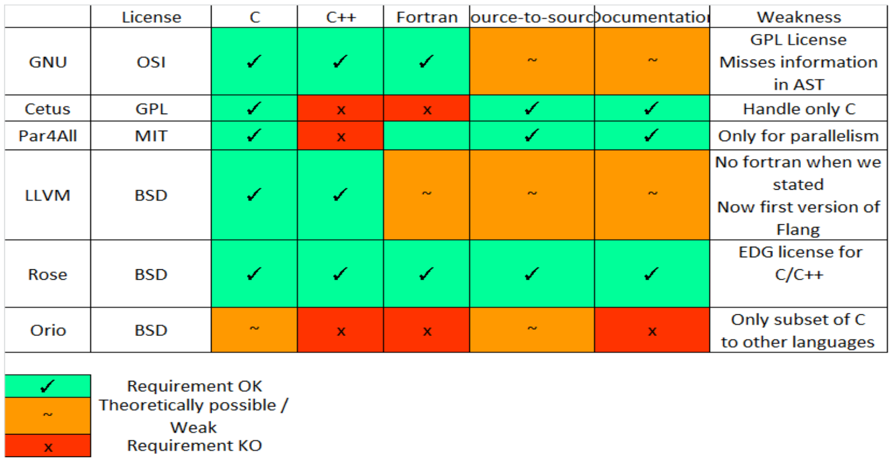
Fig. 1.1 Overview of the tool's usage.

ASSIST provides users with a simple yet flexible interface that offers two alternative approaches to specify transformations. The first one makes use of directives while the second one is based on a (Lua) script (depicted as *Transformation script*). he latest provides a means to completely hide the transformations. For example, the directive *!DIR$ MAQAO UNROLL=4* above a loop triggers the unroll (factor of 4) of its body, if applicable and by running the following command: *maqao s2s -option="apply-directives" -src=foo.f90* the transformed code can still be compiled and even reviewed by the programmer if necessary. The source code is parsed and transformed into an abstract syntax tree (AST) that ASSIST will transform accordingly into a given set of directives or a script file. Leveraging optimization opportunities is possible when feedback data from MAQAO [2,32] is available. For example, to apply the loop count transformation (described in the next section), it is possible to run *maqao s2s -vprof_xp=/path/to/vprof.csv -bin=binary*. It will use MAQAO API to search for information about loops and files to handle them and read MAQAO VPROF results to apply the loop count transformation. Available analyses are based on MAQAO CQA (code quality) and MAQAO VPROF (value profiling). Finally, the modified AST is parsed to generate a modified source file as output.

1.3.2 Compiler infrastructure

Applying transformations to a given source code requires a set of frontends. In our case we will give priority to scientific applications (HPC field), hence selecting C, C++ and Fortran languages. We want an output code that remains at source level and not in a compiler-specific intermediate representation. That is why we chose to code our transformation through the manipulation of an AST.

For all these reasons we decided to look for an existing infrastructure instead of implementing a new one.



	License	C	C++	Fortran	ource-to-sourc	ocumentatio	Weakness
GNU	OSI	✓	✓	✓	~	~	GPL License Misses information in AST
Cetus	GPL	✓	x	x	✓	✓	Handle only C
Par4All	MIT	✓	x	🟩	✓	✓	Only for parallelism
LLVM	BSD	✓	✓	~	~	~	No fortran when we stated Now first version of Flang
Rose	BSD	✓	✓	✓	✓	✓	EDG license for C/C++
Orio	BSD	~	x	x	~	x	Only subset of C to other languages

✓	Requirement OK
~	Theoretically possible / Weak
x	Requirement KO

Fig. 1.2 Constraints array

There are many compiler available infrastructures and specialized source-to-source frameworks, but only very few can satisfy our requirements. LLVM [12] is a compiler infrastructure that allows the manipulation of an AST through a library. However, it only supports C/C++ languages through Clang. Clang is very useful and easy to use to analyze an AST and add passes to the compiler but not for performing source-to-source transformations. Even if theoretically possible, it is impossible in reality due to a lack of documentation and specialized functions. Transformations are expected at the IR level. Also there is currently no production Fortran support. Very recently Flang [13] was introduced as the new Fortran frontend but it is still in its early phase of development.

Cetus [10] is a compiler infrastructure featuring source-to-source transformation of C AINSI codes only. DMS [11] is a commercial program analysis and transformation system. That is why it is not included in our comparison table.

Despite some shortfalls (refer to ASSIST's git repository) in the management of the Fortran language that we have managed to overcome, we chose Rose [27]. It is the most suitable framework given our requirements. It is the only open source and easy-to-use (i.e. documented) tool capable of manipulating the AST of C, C++ and Fortran source codes.

Figure 1.2 presents a summary of the main compiler infrastructures and specialized source-to-source frameworks. The table lists the requirements and how they are fulfilled or not. As we mentioned earlier, Rose appears to suit our constraints best.

1.3.3 Integration into MAQAO

ASSIST is a MAQAO module. That means that it has access to the MAQAO core (binary and analysis layers) and can also communicate with other MAQAO tools through an internal API. MAQAO tools deal with binary function and loop objects. Since ASSIST manipulates source code it must perform a mapping between real source lines and sources lines provided by the compiler through debug information. That way, ASSIST can establish a link between source and binary functions/loops. This implementation also allows other MAQAO tools to take advantage of ASSIST's ability to analyze and manipulate source code. That is how we extend MAQAO's ability to deal with source code.

The current implementation of ASSIST uses three MAQAO modules: LPROF for profiling (hotspots); CQA for code quality metrics (e.g. vectorization ratio); VPROF for function and loop value profiling. In this paper we only mention the features that are used by ASSIST.

1.4 Supported transformations

ASSIST features different kinds of transformations, from common ones like loop unroll to less common ones like loop and function specialization. We did not find any available tools providing such transformations. Moreover these specialization transformations have been specifically designed to be combined with the other available transformations. Block vectorization and loop count transformations are only available in ASSIST.

1.4.1 Common loop transformations

The current implementation of ASSIST supports the following common loop transformations: interchange; unroll (including full unroll); strip mine; tile. Other ones may be added in the future.

1.4.2 Constant propagation and local dead code elimination

Since we can apply multiple transformations we need a means to clean up transformed code eliminating useless chunks generated by specialization (e.g conditionals). For that purpose we implemented constant propagation and local dead code elimination.

After the constant propagation, we browse the AST to check all conditional branches. If a loop is detected with only one iteration, the loop is replaced by its body and the iteration variable replaced by its value in the whole body. ASSIST also checks "if" statements, by checking if the conditional expression is always true or false to replace the whole "if" statement by its "then" body or by its "else" body. To check if a conditional expression is always true or false, the expression is statically evaluated. If it is composed of two integers, we compare them with the corresponding operator. If it implicates a variable, ASSIST tries to trace back through previous assignment statements involving the variable to check if it ends up as a constant and if this assignment is not the result of an "if" condition or a loop. If all of the conditions are true, the variable will be considered as its value and the test continues.

1.4.3 Specialization

Specialization is the act of creating particular versions of the same code by explicitly considering specific values of one or more variables. For instance we can specialize a loop based on special values of the induction variable. Traditionally we want to handle a loop differently depending on whether it executes a low or a high number of iterations.

Specialization is not an end in itself but just a means to make optimizations happen. It is used when possible to simplify in some way a portion of code based on the knowledge of one or multiple values and their occurrences. As a consequence, the main drawback of specialization is that it can worsen performance if not used sparingly. To perform either loop or function value profiling we rely on MAQAO VPROF. Our specialization transformations can be categorized into two transformations; loop specialization and function specialization.

In the case of function specialization we will usually want to target specific value combinations. Figure 1.3 provides such an example. A new specialized function is created and the according conditionals are generated. To try to simplify the specialized code we apply our partial dead code elimination pass.

```
#pragma MAQAO SPECIALIZE(N=4,s={1,10})
void foo (int N, int* a, int* b, int s)
{
  int e = s - 10;
  if (e > 20) {
    for (int i=0; i < N; i++) {
      a[i] = b[i];
    }
  } else if (s > 10) {
     for (int i=0; i < N; i++) {
       a[i] -= b[i];
     }
  } else if (s <= 10) {
    for (int i=0; i < N; i++) {
      a[i] += b[i];
    }
}
```

(a) Before function specialization

```
void foo (int N, int* a, int* b, int s)
{
  int e = s - 10;
  if ((N==4)&&(s>0)&&(s<11)) {
   return foo_ASSIST_Ne4_sb0_11(a,b,s);
  }
  if (e > 20) {
    for (int i=0; i < N; i++) {
      a[i] = b[i];
    }
  } else if (s > 10) {
     for (int i=0; i < N; i++) {
       a[i] -= b[i];
     }
  } else if (s <= 10) {
    for (int i=0; i < N; i++) {
      a[i] += b[i];
    }
  }
}

void foo_ASSIST_Ne4_sb0_11_ei11 (int* a,
                         int* b, int s)
{
  int e = s - 10;
  for (int i=0; i < N; i++) {
    a[i] += b[i];
  }
}
```

(b) After function specialization

Fig. 1.3 Example of function specialization performed by ASSIST

It is possible to apply as many specialization directives as combinations we target. Figure 1.8 in section 1.5 is an illustration of such a case. In the current implementation specialization is limited to only integer variables.

1.4.4 Loop count transformation

We saw that loop specialization required an *a priori* knowledge of loops' bound value. This piece of information can be exploited in another way. Intel compilers offers the ability to specify a *loop count (min, max, avg)* directive. The compiler can then make that information available to its optimization passes. By default the compiler will generally generate multiple variants (e.g. scalar, SSE, AVX, etc.) of the same source loop at the binary level. However it will generate much fewer variants by considering loop count data. Helping the compiler in this way throughout the whole application can provide a significant performance gain (see section 1.5).

```
#pragma MAQAO BLOCKVECB
for (int i=0 ; i < 7; i++ ) {
  a[i] += b[i]
}
```

(a)Before

```
#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

Fig. 1.4 Example of Block Vectorization on x86_64 peformed by ASSIST

1.4.5 Block vectorization transformation

We noticed on some occasions that even when the loop bound was hard-coded the compiler would not vectorize that loop properly. We can check such cases thanks to MAQAO CQA which offers vectorization metrics. This transformation performs the following steps on a given loop: force the compiler to vectorize the loop using *SIMD* directive; prevent peeling code from being generated using *vector unaligned* directive; and adapt the number of iterations to the vector length. Figure 1.4 illustrates this transformation.

1.5 Experiments

In this section we will compare our results with the Intel compiler *pgo* mode that we will refer to as IPGO. Intel compilers are neither open source nor free, but they are available on almost all the HPC clusters and provide better performance in our tests (compared to GCC and LLVM). The main reason behind this choice of *pgo* comparison lies in the lack of FDO tools available for regular users. Our goal is not to mimic the *pgo*, rather to present a complementary approach which goes beyond 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 the Intel compiler version 17.0.4. Multiple executions (31) were performed to reach statistical stability and avoid outlier measurement data.

Also, this section presents the experimental results of the transformations offered by ASSIST based on feedback data and user insights.

Application pool

Three functional industrial applications were used to test our approach: Yales2 [9], AVBP [31] and ABINIT [14].

YALES2 is a numerical simulator of turbulent reactive flows using the Large Eddy Simulation method. It is a finite volume code for unstructured meshes, with an innovative 4th order spatial scheme for the discretization of convective and diffusive terms. It is based on the low-Mach number approximations of the Navier-Stokes equations, which solves an elliptic Poisson equation at each iteration and scales well to over 16K cores. The MPI version uses subdomain decomposition with adjustable domain size, allowing efficient cache usage. ASSIST has been tested on two of their datasets named "3D_cylinder" and "1D_COFFE". The application is written in Fortran 2003.

AVBP is parallel CFD code developped by CERFACS that solved the three-dimensional compressible Navier Stokes equations on unstructured multi-element grids. It uses third space and time Taylor Galerkin numerical schemes. The code has been ported and tested up to 200K cores with an 85% strong scaling efficiency (BG/Q) for a 200M element case (1000 elements per MPI rank). Cache coloring uses the reverse Cuthill-Mckee method. ASSIST has been tested on two representative datasets names SIMPLE (helicopter chamber demonstrator combustion simulation) and NASA (NACA blade simulation). The application is written in fortran 95.

ABINIT is a package allowing users to find the total energy charge density and electronic structure of systems made of electrons and nuclei (molecules and periodic solids) within Density Functional Theory (DFT) using pseudopotentials (or PAW atomic data) and a planewave basis. The application is developped in Fortran 90.

Impact of loop value profiling

Our first FDO optimization is based on knowing loop trip counts obtained by value profiling using MAQAO VPROF. When loops exhibit a complex control flow due to multi-versioning, knowing the trip count can help the compiler simplify the decision tree. We will refer to the loop count transformation as LCT for the remaining part of this section.

Figure 1.5 presents the speedups obtained with LCT, IPGO and the combination of both for each application/dataset. Both LCT (15%) and IPGO (14%) provide a performance gain for the Yales2 using 3D_cylinder as a dataset. The combination of both LCT and IPGO raises the gain to 19%. For the second Yales2 dataset (1D_COFFE) both endeavors only reach 5%. However, the combination does not pay off. For AVBP, running the SIMPLE data set, we observe a negligible speedup. However, for AVBP individual contributions of IPGO and ASSIST can be partially combined. IPGO provides a 10% speedup on AVBP with the NASA dataset while LCT only achieves 7%. The combination reaches 12%.

This study shows that providing the compiler with loop trip count feedback (minimum, average and maximum values) results in a performance gain. We can also observe that the combination with *pgo* can lead to a higher gain.

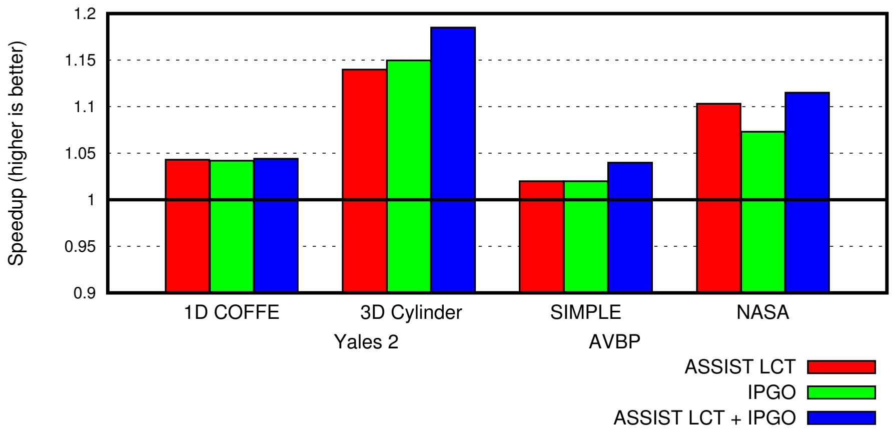


Fig. 1.5 Impact (speedup) of ASSIST LCT when compared to IPGO.

Specialization

While optimizing applications, we noticed that we often resort to function and/or loop specialization before applying other transformations. The following two examples show how coupling specialization with other transformations can provide significant performance gain.

AVBP

In this example we couple both specialization and block vectorization transformations applied to the ten most time-consuming functions. We first apply loop and function specialization separately, then we apply block vectorization on the most efficient version. We also apply the LCT and the IPGO on the original version to verify whether the compiler is able to perform better using additional guidance.

Figure 1.7 compares the speedup ratios of each version with the original one. Function and loop specialization are performed separately and presented here to show their individual impact.

We observe that block vectorization can offer a 2.6x performance gain while the loop and function specialization only achieve, at best, a speedup of 1.5x.

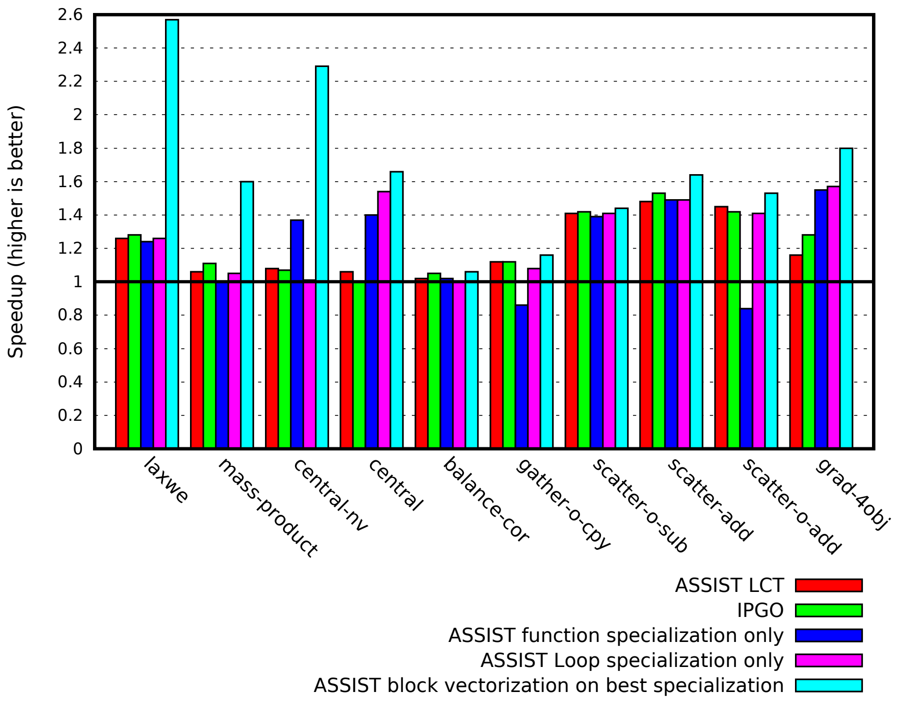


Fig. 1.6 AVBP using the SIMPLE dataset - Speedups by function before and after applying transformations with ASSIST (block vectorization, function/loop specialization, LCT) and IPGO compared with the original version (Higher is better).

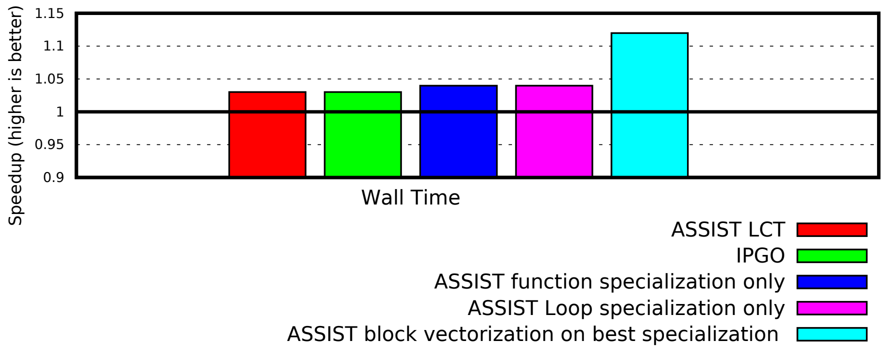


Fig. 1.7 AVBP using the SIMPLE dataset - Speedups of the wall time before and after applying transformations with ASSIST (block vectorization, function/loop specialization, LCT) and IPGO compared with the original version (Higher is better).

```
!DIR$ MAQAO SPECIALIZE(choice=1,paw_opt=3,cplex=2)
!DIR$ MAQAO SPECIALIZE(choice=1,paw_opt<3,cplex=2)
!DIR$ MAQAO SPECIALIZE(choice=1,paw_opt>3,cplex=2)
subroutine opernlb_ylm(choice,cplex,paw_opt,...)
  ...
  if (choice == 1) then
    !DIR$ MAQAO TILE_INNER_IF_SPE_choicee1=8
    do ilmn=1, nlmn
     do k=1,npw
      ztab(k) = ztab(k)+ffnl(k,1,ilmn)*cmplx(gxfacs_(1,ilmn),gxfacs_(2,ilmn),kind=dp)
     end do
    end do
  end if
  ...
end subroutine
```

(a)Before ASSIST transformations

```
SUBROUTINE opernlb_yln(...)
  IF ((choice.EQ.1).AND.(paw_opt.EQ.3).AND.(cplex.EQ.2)) then
    CALL opernlb_ylm_ASSIST_choicee1_paw_opte3_cplexe2(...)
    RETURN
  END IF
  IF ((choice.EQ.1).AND.(paw_opt.LT.3).AND.(cplex.EQ.2)) then
    CALL opernlb_ylm_ASSIST_choicee1_paw_opti3_cplexe2(...)
    RETURN
  END IF
  IF ((choice.EQ.1).AND.(paw_opt.GT.3).AND.(cplex.EQ.2)) then
    CALL opernlb_ylm_ASSIST_choicee1_paw_opts3_cplexe2(...)
    RETURN
  END IF
 ...
END SUBROUTINE
SUBROUTINE opernlb_ylm_ASSIST_choicee1_paw_opte3_cplexe2(...)
  ...
  lt_bound_npw = (npw / 8) * 8
  DO lt_var_k = 1, lt_bound_npw, 8
    DO ilmn = 1, nlmn
      DO k = lt_var_k, lt_var_k + (8 - 1)
       ztab(k) = ztab(k)+ ffnl(k,1,ilmn)* cmplx(gxfacs_(1,ilmn),gxfacs_(2,ilmn),kind=dp)
      ENDDO
    ENDDO
  ENDDO
  ...
END SUBROUTINE
SUBROUTINE opernlb_ylm_ASSIST_choicee1_paw_opti3_cplexe2(...)
...
END SUBROUTINE
SUBROUTINE opernlb_ylm_ASSIST_choicee1_paw_opts3_cplexe2(...)
...
END SUBROUTINE
```

(b) After ASSIST transformations

Fig. 1.8 ABINIT - Example of function specialization coupled with loop tiling, performed with ASSIST, for the use case Ti-256. Boxes highlight the tiling transformation of the innermost loop.

Performing only loop or function specialization may be counterproductive in some cases because of the induced complexity of the control flow if no further induced optimizations are possible.

When the compiler fails to vectorize a loop properly, the block vectorization transformation is very effective given that it explicitly exposes a simpler loop structure with no peel or tail loops to the compiler. In our case, we can evaluate the vectorization ratio of a loop using MAQAO CQA; ASSIST can automatically trigger the transformation from the CQA results by extracting several items of information, like the vectorization ratio metric, the file and the function where the loop is. The block vectorization transformations force the compiler to vectorize small loops with a small number of iterations; the compiler also fully unrolls these loops.

ABINIT

In this example, ASSIST 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 part of the code, specialization of this value simplifies address computations and vector accesses by making the stride 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 toward 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. It turns out that the dominant loop nest in the function is amenable to loop tiling. A large array is updated in its entirety inside a loop, a bad pattern for cache usage. Loop tiling make it possible to updates the array by block, 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, complicates indices, and requires a remainder loop, it is much easier and much more reliable to automate the transformation process.

Figure 1.8 shows the directives on an extract of the function, in part (a). Three specialized variants are produced for the common use cases in our reference test Ti256, by the first three lines of the figure. The critical loop nest is subsequently tiled, but only in the specialized version, by the directive immediately above the loop nest. Part (b) show extracts from the output of ASSIST. The original function now calls the specialized variants whenever the parameters are appropriate. Every conditional previously dynamically encountered is now collapsed into that one test. Below the original function, figure 1.8 also shows the new loop nest with the loop tiling transformation applied. Only 8 elements (a friendly value for a vectorizer) are computed

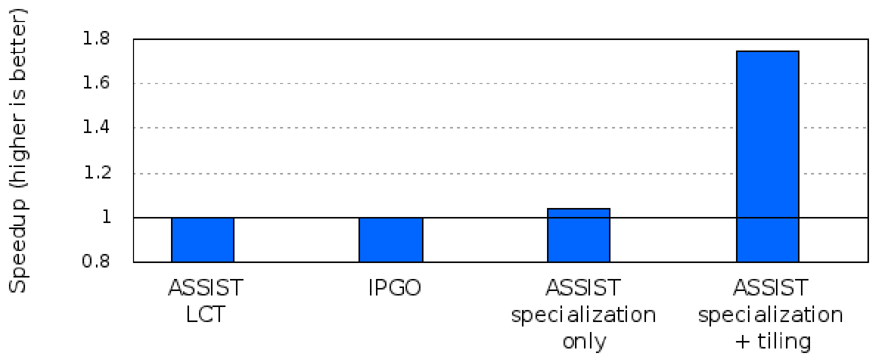


Fig. 1.9 ABINIT - Ti-256 - Speedups of IPGO, ASSIST LCT, specialized with ASSIST, specialized and tiled with ASSIST compared to the original version

in the innermost loop versus the entire array previously. An outer loop has been added which scans the entire array by block of size 8. In practice, the innermost loop is removed by the compiler, which fully unrolls and vectorizes it.

Speed up results are shown in figure 1.9. The original version is at one by definition. 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 usage of the critical loop nest. Despite the complexity of the original function, ASSIST would make it easy to apply the same transformations to other possible use cases of the function for other test cases of the ABINIT code.

1.6 Related Work

The originality of the approach presented in this paper lies in the combination of both source-to-source transformations using annotations and FDO approaches. More precisely feedback data drives source-to-source transformations to achieve both productivity and performance.

Orio [16] is the closest tool and approach to ASSIST. We share the same goals, namely improving productivity and performance using annotations at source level as well as being able to handle architecture-specific/independent code optimizations. However, they use empirical performance tuning to achieve better performance. This implies generating multiple variants and evaluating their cost. Our approach opts for a cheaper and more straightforward path using FDO. Our approach encompasses static and dynamic analyses. This means that we can assess the quality of the code generated by the compiler (using MAQAO CQA [30]) and get execution behavior metrics. CHiLL [7] is a framework that provides loop level transformations and also uses empirical optimizations. It targets compilers and not regular developers. Xevtgen [28] goes a step further when considering source-to-source transformations. It allows application developers to define their own transformations using a dummy Fortran syntax coupled with directives. From our own experience in helping developers optimize their code, we can claim it is dangerous to assume they will be willing to invest time and ressourses to write their own transformations, even if the interface is based on a well-known language such as Fortran. For this particular reason, we have tried to provide as many predefined transformations as possible. Also, plenty of Domain Specific Languages or frameworks are available for performing source-to-source transformations, i.e: [8, 11, 21, 26, 36]. Some also implement parallel transformations [1, 4, 18, 24, 25, 33].

For FDO, the related work analysis is straightforward: there are very few tools and the main goal is to achieve performance. From what we encountered during our research, the only available tools implementing FDO are compilers, with PGO (e.g. Intel, GCC, LLVM), and AutoFDO (e.g. Intel [17], GCC [15] and LLVM) modes. AutoFDO [6] is also 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 are injected early in the intermediate representation of the compiler so that all the optimization passes can take advantage of them. Our approach aims to help modern compilers by not injecting data using a specific format, but rather at source level. From a performance point of view, both approaches are complementary. During our work, we also came across Aestimo [23], an FDO research evaluation tool that can be coupled with the Open Research Compiler [34]. However it does not pursue the same goals.

1.7 Conclusion and future work

ASSIST is an open source tool that was developed with the aim of providing assistance to application programmers in order to achieve better productivity and code performance. We have shown the effectiveness of our approach when dealing with industrial applications by using either static and dynamic feedback data, or user guidance.

The tool presented in this article provides the foundation for an autotuning tool. As future work we plan to harness all the available dynamic analyses existing in MAQAO including those using hardware counters to perform and automate more optimizations.

1.8 Acknowledgements

We would like to thank Gabriel Staffelbach (CERFACS) for providing our laboratory with the AVBP application, as well as Ghislain Lartigue and Vincent Moureau (CORIA) for providing us with YALES2.

This work has been carried out by the Li-PaRAD laboratory, PeXL and the Exascale Computing Research laboratory, with the support of CEA, Intel, UVSQ. Intel granted us dedicted access to a Skylake SP machine on which the experiments were run.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the CEA, Intel, or UVSQ.

References

1. Advisor. https://software.intel.com/en-us/intel-advisor-xe.
2. Denis Barthou, Andres Charif Rubial, William Jalby, Souad Koliai, and Cedric Valensi. Performance tuning of x86 openmp codes with maqao. In *Parallel Tools Workshop*, pages 95—113, Desden, Germany, September 2009. Springer-Verlag.
3. Zakaria Bendifallah, William Jalby, José Noudohouenou, Emmanuel Oseret, Vincent Palomares, and Andres Charif Rubial. *PAMDA: Performance Assessment Using MAQAO Toolset and Differential Analysis*, pages 107–127. Springer International Publishing, Cham, 2014.
4. U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *ACM SIGPLAN Notices*, page 101113. ACM, 2008.
5. A. S. Charif-Rubial, D. Barthou, C. Valensi, S. Shende, A. Malony, and W. Jalby. Mil: A language to build program analysis tools through static binary instrumentation. In *20th Annual International Conference on High Performance Computing*, pages 206–215, Dec 2013.
6. Dehao Chen, David Xinliang Li, and Tipp Moseley. Autofdo: Automatic feedback-directed optimization for warehouse-scale applications. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, CGO '16, pages 12–23, New York, NY, USA, 2016. ACM.
7. Jacqueline Chame Chun Chen and Mary Hall. Chill: A framework for composing high-level loop transformations. June 2008.
8. James R. Cordy. Source transformation, analysis and generation in txl. In *Proceedings of the 2006 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, PEPM '06, pages 1–11, New York, NY, USA, 2006. ACM.
9. Coria. http://www.coria-cfd.fr/index.php/YALES2.
10. Dave and al. Cetus: A source-to-source compiler infrastructure for multicores. In *Computer*, pages 36—42, December 2009.
11. Chris Lattner et Vikram Adve. Dms/spl reg: program transformations for practical scalable software evolution. In *Software Engineering, ICSE 2004. Proceedings. 26th International Conference on*, page 625634. IEEE, 2004.
12. Chris Lattner et Vikram Adve. Llvm a compilation framework for lifelong program analysis and transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, page 75. IEEE Computer, 2004.
13. flang. https://github.com/llvm-flang/flang.

14. X. Gonze and al. Abinit: First-principles approach to material and nanosystem properties. In *Computer Physics Communications*, pages 2582–2615. Elsevier, 2009.
15. Google. https://github.com/google/autofdo.
16. A. Hartono, B. Norris, and P. Sadayappan. Annotation-based empirical performance tuning using orio. In *2009 IEEE International Symposium on Parallel Distributed Processing*, pages 1–11, May 2009.
17. Intel. https://github.com/google/autofdo.
18. Irigoin and al. Interprocedural analyses forprogramming environments. In *Workshop on Evironments and Tools For Parallel Scientifc Computing*, Saint-Hilaire du Touvier, France, August 1992.
19. Andres S. CHARIF RUBIAL Jean-Baptiste Lereste. https://www.maqao.org/release/MAQAO.Tutorial.LProf.v1.pdf.
20. Souad Koliaï, Zakaria Bendifallah, Mathieu Tribalat, Cédric Valensi, Jean-Thomas Acquaviva, and William Jalby. Quantifying performance bottleneck cost through differential analysis. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 263–272, New York, NY, USA, 2013. ACM.
21. R. vermaas 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, May 2008.
22. Diego Novillo. Samplepgo: The power of profile guided optimizations without the usability burden. In *Proceedings of the 2014 LLVM Compiler Infrastructure in HPC*, LLVM-HPC '14, pages 22–28, Piscataway, NJ, USA, 2014. IEEE Press.
23. J.N. Amaral P. Berube. Aestimo: a feedback-directed optimization evaluation tool. Piscataway, NJ, USA, 2006. IEEE.
24. Marek Palkowski and Wlodzimierz Bielecki. *TRACO Parallelizing Compiler*, pages 409–421. Springer International Publishing, Cham, 2015.
25. Paraformance. http://paraformance.weebly.com/.
26. Jurgen Vinju Paul Klint, Tijs van der Storm. Rascal a domain specific language for source code analysis ad manipulation. In *IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 168–177. IEEE Computer Society, 2009.
27. Quinlan and al. Rose: Compiler support for object-oriented framework. In *Parallel Processing Letters*, pages 215—226, Lawrence Livermore National Laboratory, Livermore, CA, USA, October 2000. World Scientific.
28. Hiroyuki Takizawa Reiji Suda and Shoichi Hirasawa. Xevtgen: Fortran code transformer generator for high performance scienti c codes. In *International Journal of Networking and Computing*, pages 263—289, July 2016.
29. Francois Bodin Romain Dolbeau, Stphane Bihan. Hmpp: A hybrid multi-core parallel programming environment. In *Workshop on general purpose processing on graphics processing units (GPGPU 2007)*, volume 28, 2007.
30. Andres Charif Rubial, Emmanuel Oseret, Jose Noudohouenou, William Jalby, and Ghislain Lartigue. CQA: A code quality analyzer tool at binary level. In *HiPC*, pages 1–10. IEEE Computer Society, 2014.
31. M. Rudgyard T. Schonfeld. Steady and unsteady flow simulationsusing the hybrid flow solver avbp. In *AIAA Journal*, pages 1378–1385. AIAA ARC, 1999.
32. MAQAO toolsuite. http://www.maqao.org.
33. S. Verdoolaege and al. Polyhedral parallel code generation for cuda. In *ACM Trans. Architec. Code Optim.* ACM, January 2013.
34. Chengyong Wu, Ruiqi Lian, Junchao Zhang, Roy Ju, Sun Chan, Lixia Liu, Xiaobing Feng, and Zhaoqing Zhang. *An Overview of the Open Research Compiler*, pages 17–31. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.
35. X. Xiao, S. Hirasawa, H. Takizawa, and H. Kobayashi. An approach to customization of compiler directives for application-specific code transformations. In *2014 IEEE 8th International Symposium on Embedded Multicore/Manycore SoCs*, pages 99–106, Sept 2014.

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