This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2020.3001257, IEEE Transactions on Software Engineering

Pegasus: Performance Engineering for Software Applications Targeting HPC Systems

Pedro Pinto, João Bispo, João M.P. Cardoso, *Senior Member, IEEE*, Jorge G. Barbosa, *Member, IEEE*, Davide Gadioli, Gianluca Palermo, *Member, IEEE*, Jan Martinovič, Martin Golasowski, Kateřina Slaninová, Radim Cmar, Cristina Silvano, *Fellow, IEEE*

Abstract—Developing and optimizing software applications for high performance and energy efficiency is a very challenging task, even when considering a single target machine. For instance, optimizing for multicore-based computing systems requires in-depth knowledge about programming languages, application programming interfaces (APIs), compilers, performance tuning tools, and computer architecture and organization. Many of the tasks of performance engineering methodologies require manual efforts and the use of different tools not always part of an integrated toolchain. This paper presents Pegasus, a performance engineering approach supported by a framework that consists of a source-to-source compiler, controlled and guided by strategies programmed in a Domain-Specific Language, and an autotuner. Pegasus is a holistic and versatile approach spanning various decision layers composing the software stack, and exploiting the system capabilities and workloads effectively through the use of runtime autotuning. The Pegasus approach helps developers by automating tasks regarding the efficient implementation of software applications in multicore computing systems. These tasks focus on application analysis, profiling, code transformations, and the integration of runtime autotuning. Pegasus allows developers to program their strategies or to automatically apply existing strategies to software applications in order to ensure the compliance of non-functional requirements, such as performance and energy efficiency. We show how to apply Pegasus and demonstrate its applicability and effectiveness in a complex case study, which includes tasks from a smart navigation system.

1 Introduction

PERFORMANCE and energy consumption are increasingly essential non-functional requirements (NFRs) in software engineering. To achieve performance and energy efficiency goals, software developers require a deep understanding of both the problem at hand and the target computer architecture (see, e.g., Cardoso et al. [1]). Moreover, software developers have to consider a multitude of programming models and languages, tools, and heterogeneous architectures and systems, which increases the development complexity when dealing with those NFRs. Although the number of software applications needing high performance and energy efficiency is increasing, only specialized developers master this necessary knowledge. Thus, methodologies and tools to assist both specialized and typical developers are of paramount importance when targeting high-performance computing (HPC) systems.

The need to optimize applications and to take advantage of the current and future HPC systems [2], especially based on the heterogeneous computing power capability, is fully recognized as an important contribution to achieve energy efficiency goals [3]. Such optimizations may involve compiler optimizations, code transformations, parallelization and specialization [4, 5, 6]. Typically, to satisfy performance and energy or power consumption requirements, software applications are given to tuning experts and recently to performance engineers, who need to dig in the refactoring space and select suitable code transformations.

Software development does not start with a focus on the satisfaction of performance and energy or power consumption requirements, which could even be counter-productive in some cases. The typical methodology followed by expert developers and performance engineers for improving software applications in terms of execution time and energy or power consumption requires several tasks. Commonly, developers analyze the application (e.g., with profiling), make decisions regarding code transformations, tuning of parameters, and compiler options. In more sophisticated cases, developers may consider the inclusion of runtime autotuning strategies [7], used to adapt applications to the dynamic conditions of the execution environment. Fig. 1 shows a typical performance engineering methodology flow for HPC and consisting of the following main tasks:

- **Analysis and Profiling:** Incremental analysis and profiling of the software application and impact of NFRs. This analysis can rely on static and dynamic information and may involve "what-if" analysis and design-space exploration (DSE).
- **Strategy Selection and Development:** Selection of strategies to target NFRs. With the knowledge acquired by the analysis, developers can decide to apply strategies from a catalog (e.g., loop transfor-

- *Pedro Pinto, João Bispo, João M. P. Cardoso and Jorge G. Barbosa are with the Department of Informatics Engineering, Faculty of Engineering, University of Porto, Porto, Portugal.*
  Email: {p.pinto, jbispo, jmpc, jbarbosa}@fe.up.pt
- *Davide Gadioli, Gianluca Palermo and Cristina Silvano are with the Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano, Milano, Italy.*
  Email: {davide.gadioli, gianluca.palermo, cristina.silvano}@polimi.it
- *Jan Martinovič, Martin Golasowski and Kateřina Slaninová are with IT4Innovations, VSB, Technical University of Ostrava, Ostrava, Czech Republic.*
  Email: {jan.martinovic, martin.golasowski, katerina.slaninova}@vsb.cz
- *Radim Cmar is with Sygic, Bratislava, Slovakia.*
  Email: rcmar@sygic.com

mations and automatic parallelization) or apply custom strategies. At this stage, developers also make decisions about whether and how to include runtime autotuning;

- **Autotuner Integration:** Integration of runtime autotuning and other libraries, as well as generation and selection of the configurations to be used at runtime;
- **Application Deployment:** Ultimately, developers generate the final version of the application code and deploy it.

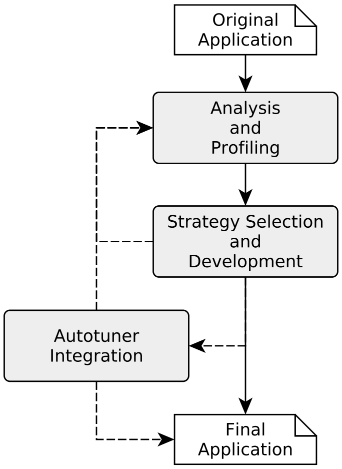


Fig. 1. Main tasks of a typical performance engineering methodology.

All the tasks involved in Fig. 1 rely on multiple tools, mostly selected based on the knowledge and familiarity of the developer or performance engineer, with the help of hard manual work efforts. The integration of the tools in a single framework is usually missing and would need high-levels of flexibility to adopt specific tools in each stage of the methodology. Thus, specific actions are done manually due to the lack of adequate tools or to the lack of integration with other tools.

Bearing in mind some of these issues as a way to contribute to the automation of the previously introduced performance engineering methodology, we have adopted the main concepts of an approach [8, 9, 10] inspired on Aspect-Oriented Programming (AOP) [11], originally proposed in the context of embedded systems [9, 12], and further developed in the context of HPC applications [13, 14], to introduce in this paper the Pegasus approach. Pegasus relies on previously developed components and on particular enhancements to contribute to the automation of the methodology presented in Fig. 1. In particular, we use the LARA DSL and its associated libraries [15] to assist developers and performance engineers when developing and tuning C/C++ applications, the Clava[1] C/C++ source-to-source compiler, and the mARGOt runtime autotuner [16]. The LARA language was originally developed to assist developers when targeting multicore embedded devices consisting of reconfigurable hardware. Initially, there was a focus on instrumentation to identify critical regions and guide mapping, hardware/software partitioning, and word-length optimization. Clava, its supporting libraries, and the current version of the mARGOt autotuner were initially proposed in ANTAREX[2] project and their ultimate versions are core components of the framework presented in this paper.

LARA allows developers to program strategies ("recipes") and automatically apply them to software applications using a concept similar to AOP weaving [11]. AOP is a programming paradigm aimed at increasing program modularity by encapsulating code related to crosscutting concerns (such as logging, profiling, and autotuning) into separate entities called aspects that are then woven into the original application. One of our goals is to maintain the application software code mainly concerned with its business logic and separated from the code related to NFRs as much as possible, by generating the modified software application automatically.

This paper introduces Pegasus, an integrated approach that can automatize the methodology presented in Fig. 1 by relying on the previously described components. Pegasus contributes to a more systematic process, which is helpful to assist developers and performance engineers, dealing with execution time and energy or power consumption requirements. We show examples of recurring concerns arising from the tasks of the presented HPC methodology and how developers and performance engineers can use Pegasus to program custom strategies to address those requirements. Furthermore, we show the use of the Pegasus approach to assist various performance engineering stages and tasks in the context of a smart future navigation system running in an HPC platform.

Overall, the main contributions of this paper are the following:

- A systematic approach to support developers and performance engineers when dealing with execution time and energy or power consumption requirements;
- An integrated and smooth use of runtime autotuning, including the synthesis and automatic integration of state-of-the-art runtime autotuning schemes;
- An evaluation of the approach with a large-scale and high-computing-complexity case study, an industrial prototype for a smart and future navigation system to be run on an HPC system.

The remainder of this paper is organized as follows. Section 2 describes the primary motivation for the proposed Pegasus approach. Section 3 presents the approach and its main components. In Section 4, we describe some representative use cases and the use of Pegasus. Section 5 presents the case study and describes how to apply the performance engineering methodology using Pegasus. Section 6 shows the experimental results and an evaluation of the Pegasus approach to the case study. Section 7 reviews the related work, while Section 8 concludes the paper and presents future work.

2 Motivation

Performance engineering for HPC applications typically involves the tasks shown in Fig. 1. These tasks can be

[1] Clava source code: https://github.com/specs-feup/clava

[2] For more information, please see: http://antarex-project.eu/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2020.3001257, IEEE Transactions on Software Engineering

seen as sequential phases, but are generally iterative. In practice, developers perform multiple cycles of analysis, development, and integration to fine-tune an application to the non-functional requirements.

All tasks require analysis of the source code of the software application, selection of points of interest, and instrumentation or transformations of the code. In the first task, *Analysis and Profiling*, these steps are performed to gather knowledge about the application, while in the other two tasks, *Strategy Selection and Development* and *Autotuner Integration*, the application is modified to meet the desired goals and requirements. For example, in *Analysis and Profiling*, developers may need to make code changes that are later discarded, since they might be applied only to collect runtime characteristics of the application.

A framework to deal with the presented methodology stages needs to have enough flexibility to support the automation of several actions. These actions range from the analysis of software code (e.g., to acquire static information or to identify bugs in the application [17]), the instrumentation of the applications (e.g., to acquire dynamic information), the modifications of code, to the integration and synthesis of runtime autotuning schemes.

One of the core actions in the *Strategy Selection and Development* task is code refactoring, also known as code restructuring [18, 19] and code transformation. Code refactoring was originally recognized as beneficial for improving the quality of software, e.g., regarding robustness, extensibility, reusability, and performance [20]. More recently, it has been used for reducing energy consumption and for the parallelization [21]. In many cases, users do not perform code refactoring due to their unawareness of tools (as mentioned by Murphy-Hill et al. [22]), or the lack of time and the risk associated with transforming the code [23]. These reasons apply mainly when dealing with code quality goals, such as maintainability, extensibility, and reusability. However, when the goals involve execution time and energy or power consumption, the causes are not only the users' unawareness of tools, but also the lack of tools, the lack of knowledge regarding the vast portfolio of code transformations, the complexity to devise sequences of transformations, and the lack of an easy way to know the impact of those transformations. The fact that many HPC application developers are domain experts, but neither computer scientists nor performance engineers, further aggravates this problem. Therefore, it is essential to provide tools to help users to address these problems and to apply code refactoring, towards reaching maximum peak performance.

Herein, we demonstrate the application of some actions of each task of the methodology to a simple matrix multiplication code, a well-known and straightforward example, with the relevant code excerpt shown in Fig. 2. Matrix multiplication has been intensively studied [24, 25], and there exist very optimized HPC implementations. This example, however, is simple enough to follow and to show several tasks to be done.

One of the first actions for performance analysis of an application is profiling. For that, one can use GNU *gprof* [26], Linux *perf*[3] or tools provided by Valgrind [27], e.g., for cache and call-graph profiling. The profiling reveals meaningful information, e.g., where the execution of the application spends most of its time (code regions or functions known as hotspots). We note, however, that other analyses might be involved, and there are tools, such as Vampir [28], that can help on performance analysis of parallel applications.

```
// ...
template< typename T >
void matrix_mult(const vector<T>& A , const vector<T>& B,
    vector<T>& C, const int N, const int M, const int K) {
    // ...
    for(int i=0; i<N; i++) {
        for(int l=0; l < M; l++) {
            for(int j=0; j < K; j++) {
                C[K*i + j] += A[M*i+l]*B[K*l+j];
            }
        }
    }
}
// main function here...
```

Fig. 2. Main parts of the original matrix multiplication code.

Let us assume that the profiling information reveals that the matrix multiplication function, `matrix_mult`, accounts for most application's execution time, and thus it is the function where developers shall focus the first optimization efforts.

To assess the impact of code transformations or to have direct measurements of code regions, it is common to instrument the application to measure time and energy around a region of interest. In this example, we use standard C++ libraries to measure the time elapsed around the call to the `matrix_mult` function. To measure energy consumption, we rely on a library that makes use of RAPL [29]. Fig. 3 presents the resulting code.

```
#include <iostream>
#include <chrono>
// ...
int main() {
    // ...
    auto e0 = rapl_energy();
    auto t0 = chrono::high_resolution_clock::now();

    matrix_mult(A, B, C, N, M, K);

    auto t1 = chrono::high_resolution_clock::now();
    auto e1 = rapl_energy();
    cout << (e1-e0)  << "uJ" << endl;
    auto d = t1 - t0;
    auto d_ms =
        chrono::duration_cast<chrono::milliseconds>(d);
    cout << d_ms.count() << "ms" << endl;
    // ...
}
```

Fig. 3. The function call to the kernel in the main is instrumented for measuring execution time and energy consumption.

Now, we can easily measure the execution time of the original and any newly generated code version and compare those versions to evaluate the impact of possible optimizations. The output of the execution reports the time spent in each kernel call, in addition to the original information, as seen in Fig. 4.

At this stage, it is common to analyze the code of the application (mostly the code of the hotspots) and to select code optimizations that can improve performance.

[3] https://perf.wiki.kernel.org/index.php/

```
#0 C[0][0] = 128.153 [512x512] X [512x512]
2.36459e+06 uJ
94 ms
```

Fig. 4. Part of the output of the program, including the timing and energy consumption information for each function call.

For instance, in order to reduce the execution time, we applied loop tiling [30] to the loops of the function. Loop tiling can provide better locality and reduce cache misses and, therefore, reduce execution time and energy or power consumption. Another possibility can be the use of loop interchange [30], which requires an analysis of the iteration space and access patterns in order to select the loops to interchange. In this example, we applied loop tiling to the three loops in the critical loop nest of the function. Fig. 5 shows an excerpt from the resulting code.

```
// ...
template< typename T >
void matrix_mult_tiling(const vector<T>& A ,
  const vector<T>& B, vector<T>& C,
  const int N, const int M, const int K) {

  const int BS1 = 32;
  const int BS2 = 32;
  const int BS3 = 32;
  // ...

  for(int i2=0; i2<N; i2 += BS1) {
    for(int l2=0; l2<M; l2 += BS2) {
      for(int j2=0; j2<K; j2 += BS3) {
        for(int i=i2; i< min(N, i2+BS1); i++) {
          for(int l=l2; l< min(M, l2+BS2); l++) {
            for(int j=j2; j< min(K, j2+BS3); j++) {
              C[K*i + j] += A[M*i+l]*B[K*l+j];
            }
          }
        }
      }
    }
  }
}
// ...
```

Fig. 5. The main kernel transformed with loop tiling.

The choice of the optimal tile size is not trivial and depends on factors that might be unknown at the time we improve the code. For instance, the memory organization and sizes of the caches of the target machine play an important role, requiring the developer that tunes the code to know the target machine beforehand. Another factor that affects the choice of the tile size is the size and shape of the matrices used.

The next step is to measure the execution time and the energy consumption for different tile and input matrices sizes. It is common that at this stage, developers use design-space exploration (DSE) tools (see, e.g., [31]) to evaluate the different configuration settings. However, it is also not uncommon that developers perform this exploration manually via code modifications, sometimes incurring lengthy and error-prone development efforts.

We performed the exploration of tile and input matrices sizes for two different machines, *A* and *B*, to illustrate how different architectures affect the choice of tile size. Table 1 illustrates the results of this exploration for machine *A* and presents the speedups of the code versions with loop tiling over the original version (i.e., without loop tiling). Here, developers may need to execute several times (five runs in this example) each version of the application and report average execution time and energy consumption. We note that, albeit not presented, the energy consumption of these versions followed the speedup trends.

TABLE 1
Speedups for machine *A* over the original application (without loop tiling) for the explored combinations of matrix size (rows) and tile size (columns). Results for the best tile sizes for each matrix size are highlighted in **bold**.

Matrix Size	Tile Size				
	64	128	256	512	1024
512	0.77	0.80	**0.91**	-	-
1024	0.68	0.76	0.89	**0.94**	-
2048	1.25	1.52	1.80	1.97	**2.08**
4096	1.30	1.60	1.84	**2.06**	1.03
8192	1.30	1.58	**1.83**	0.98	0.99

These results show the importance of considering both tile and matrix sizes. In some of the cases, namely for matrices of size 512, loop tiling with the explored tile sizes does not bring any improvement in execution time. The results across a row illustrate how the choice of tile size affects the performance for a particular matrix size. Those results also show how the cache sizes and organization affect the choice of this parameter. For instance, the row for matrix size 8192 presents slowdowns for large tile sizes (0.98× for 512), and speedups for smaller tile sizes (1.83× for 256).

The target machine needs to be taken into account to assess the impact on the performance of the chosen tile sizes. For instance, while for machine *A*, the best tile sizes are {256, 512, 1024, 512, 256} for each of the five matrix sizes, for machine *B* the best tile sizes are {256, 256, 512, 512, 256}.

On the other hand, the results across a column show that developers should also consider the matrix size. For instance, the column for tile size 64 shows slowdowns when used for smaller matrices (0.68× for 1024), but speedups when used for larger matrices (1.30× for 8192).

Although these experiments illustrate the need for exploration and the kind of work needed to achieve this analysis, they consist of an elementary and limited exploration. Typically, developers may need to test a larger set of values and to consider all the parameters (variables) separately. For instance, in our exploration example, the tile size variables, `BS1`, `BS2` and `BS3`, have always an equal value. Similarly, the variables with the sizes of the matrices, `M`, `N`, and `K`, have always an equal value, i.e., we only tested the multiplication of squared matrices. The shape of the matrix may also impact the choice of tile size, which for simplification, we did not take into account.

One critical optimization consists of parallelizing the application, e.g., via OpenMP directives [32]. We extended the previous exploration for the matrix size of 2048 to test the effect of different tile sizes and different numbers of threads in a parallel version. The best result for a serial application, a tile size of 1024, does not scale when using more than two threads. This was expected as each of the two threads deals with chunks of data with the same size as the tile, i.e., 1024 elements. This pattern is also observed for tiles of size 512 and 4 threads, of size 256 and 8 threads, of size 128 and 16 threads, and of size 64 and 32 threads. The exploration of the number of threads in {1, 2, 4, 8, 16, 32} showed that the fastest execution time is achieved with the tile size of 128 and using 32 threads. Additional exploration parameters could be the scheduling policy (highly dependent on the problem), and the distribution of threads on the machine (highly dependent on the architecture).

Thus, for thorough exploration, developers may have to deal with large design spaces, thus requiring sophisticated DSE schemes. As most of the strategies involve code instrumentation and configuration, and there is a vast design space to consider, manually changing the application to support and perform the exploration can be unfeasible, time-consuming, and prone to errors.

In specific scenarios, a runtime selection of a particular configuration is more advantageous. For instance, when the best configuration depends on the input data used or on the target machine (as shown before), developers may have to enhance the application with the capability to postpone configuration decisions to runtime. In this case, the solution involves the integration of a runtime autotuner.

In the matrix multiplication case, the use of runtime autotuning can postpone the choice of the tile sizes to execution time. However, even in this case, some offline exploration might be needed to generate a knowledge-base for the autotuner. For instance, considering execution time and energy consumption metrics, a Pareto frontier (see, e.g., Li and Yao [33]), would enable the autotuner to control this trade-off by choosing the values of the variables.

We parameterized the matrix multiplication function with the tile sizes, and we inserted the autotuner code to choose the tile sizes immediately before the call. The decision takes into account the current running conditions (as measured by the internal monitors of the autotuner) and the sizes of the input matrices. Fig. 6 shows an excerpt of a version of the application that uses mARGOt [16] to provide this online adaptation. The tile sizes became a parameter of the kernel, and the autotuner sets their value before the function call with the `update` call to the mARGOt interface. The autotuner receives the sizes of the matrices, `N`, `M`, `K`, as inputs and sets the values of `BS1`, `BS2` and `BS3` right before the call site. The other calls to mARGOt start and stop its internal monitors, which in this case, keep track of the execution time.

With this simple matrix multiplication code, we have shown several techniques typically used by performance engineers. This example illustrates the type of work needed and how it can scale, but it also shows that even for straightforward cases, there is a need for an integrated methodology to support the application developer.

The next section describes the Pegasus approach and associated tool flow to semi-automate the tasks of the proposed performance engineering methodology. Those tasks include analysis, instrumentation, code transformations, design-space exploration, and integration of a runtime autotuner.

```
#include <margot.hpp>
//...
template <typename T>
void matrix_mult_tiling(vector<T> const& A,
  vector<T> const& B, vector<T>& C,
  int const N, int const M, int const K,
  int const BS1, int const BS2, int const BS3) {
// ...
}

int main() {
  margot::init();
  // ...
  int BS1, BS2, BS3;
  // ...
  if(margot::matmul::update(BS1, BS2, BS3, N, M, K)) {
    margot::matmul::manager.configuration_applied();
  }
  margot::matmul::start_monitor();
  matrix_mult_tiling(A, B, C, N, M, K, BS1, BS2, BS3);
  margot::matmul::stop_monitor();
// ...
}
```

Fig. 6. The call to the matrix multiplication function was surrounded with autotuner code that chooses the best tile size from a set of pre-fixed tile sizes for the current execution context.

3 Performance Engineering Approach

The Pegasus approach uses a framework composed of Clava, a source-to-source compiler, LARA [9, 10], the language used to program strategies that are automatically applied in the performance engineering tasks, and mARGOt [16], a runtime autotuner. Pegasus covers the tasks presented in Section 1 with the following steps:

1) **Analysis and Profiling:** Analysis of the application code, and profiling of its runtime behavior and impact of certain transformations, parameter values, and algorithms;
2) **Strategy Selection and Development:** Selection of code transformations, compiler optimizations, and decisions regarding the analysis in the previous step, including the development of new and custom transformations;
3) **Autotuner Optimization:** Generation of the Knowledge Database and identification of Pareto frontiers for the generation of the autotuning model and synthesis of the runtime autotuner;
4) **Autotuner Integration:** Insertion of the runtime autotuner in the application code;
5) **Application deployment**.

The analysis can be either based on looking at the current state of the application or based on a "what-if" analysis. The former tries to understand how an application is currently working and if we can take advantage of its characteristics and inputs (through profile-guided optimizations). These analyses include timing and energy profiling of the application to find hotspots (i.e., code regions of the application with the most significant contribution to a given metric) as well as input frequency analysis, e.g., used to guide memoization techniques [34]. The latter type of analysis relies on LARA strategies to "poke and probe" the application and to test what happens if a parameter or algorithm is changed. A developer can perform such an analysis through ad hoc LARA strategies or, more systematically, by relying on

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2020.3001257, IEEE Transactions on Software Engineering

exploration libraries provided by Clava to perform design-space exploration and measure different metrics of interest. For instance, these strategies can test the impact of data type conversion between half-, single-, and double-precision floating-point types, or the impact of changing the number of threads of an OpenMP program.

The optimization and integration phases build on the results of the analysis. These phases are often part of a loop, in which we come back to the analysis after transforming and optimizing critical parts of the application and including other components.

Our approach relies on a tool flow that uses Clava and LARA throughout all the steps, as shown in Fig. 7. They are used to define strategies for all the steps, from analysis to optimization and integration of other components.

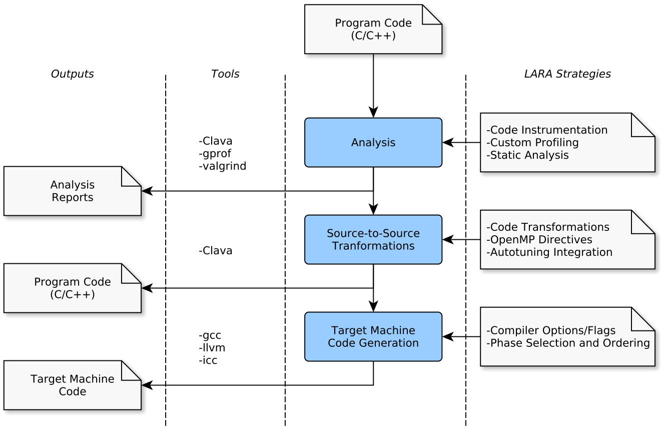


Fig. 7. The Clava+LARA tool flow.

3.1 The LARA Language

The LARA [9, 10] language provides several constructs for capturing, querying, and modifying the source elements of a target application. Furthermore, it is possible to use arbitrary JavaScript to provide general-purpose computation. The most important LARA constructs can be summarized as follows:

- `aspectdef` marks the beginning of an aspect. The aspect is the main modular unit of the LARA language.
- `select` allows to query elements in the code (e.g., `function`, `loop`) that we want to analyze or transform. This selection is hierarchical, i.e., `select function.loop end`) selects all the loops inside all the functions in the code.
- The `apply` block iterates over all the elements of the previous selection. Each particular point in the code, herein referred to as *join point*, can be accessed inside the `apply` block by prefixing $ to the name of the join point (e.g., `$loop`). Each join point has a set of *attributes*, which can be accessed, and a set of *actions*, which can be used to transform the code.
- The `condition` block can be used to filter join points over a join point selection.

3.2 The Clava Source-to-Source Compiler

We base our approach on the idea that specific tasks and application requirements (e.g., target-dependent optimizations, adaptivity behavior, and concerns) can be specified separately from the source code that defines the functionality of the program. Developers and performance engineers can express those requirements as reusable strategies written in a DSL and applied as a compilation step. To implement this approach for C/C++ programs, we developed the Clava source-to-source compiler, that applies source code analysis and transformation strategies described in the LARA language.

Fig. 8 shows a block diagram of the Clava+LARA framework, which is composed by three main parts: 1) the *LARA Framework*; 2) the *Clava Weaver engine*; and 3) the *C/C++ Frontend*.

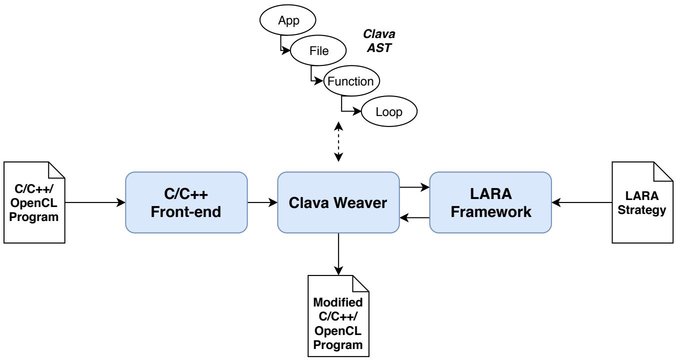


Fig. 8. Block diagram of the Clava+LARA framework.

The *LARA Framework* compiles and executes the LARA strategies defined in the input aspect files, instructing the weaver on which code elements to select, which information to query, and which actions to perform.

The *C/C++ Frontend* transforms the source code of the input application into an abstract representation that can be manipulated by the Clava Weaver engine. The Frontend was implemented using the Clang compiler[4], which is used to parse the code and build an Abstract Syntax Tree (AST) that is manipulated by the Clava Weaver engine. This AST closely resembles the internal AST of Clang, with modifications and extensions that allow AST-based transformations, and the capability of generating source code that is, as much as possible, similar to the original.

The *Clava Weaver* engine is responsible for maintaining an updated internal representation of the application source code, initially generated by the C/C++ Frontend, which is manipulated according to the execution of LARA strategies. At the end of the execution, it generates the woven application source code from the AST.

Current Clava libraries allow users to enhance their applications with, e.g., memoization and autotuning capabilities. These libraries can be imported and used in LARA and deal with the generation of code and configuration files that are needed for those libraries to work.

The C preprocessor (CPP) is commonly used by developers in HPC scenarios, e.g., for targeting different archi-

[4] Clang: a C language family frontend for LLVM. For more information, please visit http://clang.llvm.org/

tectures. Clava interacts with CPP by obtaining an AST after the code has been transformed by CPP, as Clang invokes CPP before parsing the source code. Thus, source code transformations are applied later in the build process after the CPP has resolved all definitions and conditional statements.

3.3 Source-to-Source Transformations

Source-to-source transformations are a crucial part of the performance engineering methodology, and Pegasus supports them through Clava. There are two main reasons to change the application code. The first is to improve the performance of an application, which can be done directly, e.g., by applying loop transformations, or indirectly, e.g., by introducing specialized versions of critical functions and mechanisms to decide which versions to run depending on the current context. The second reason is to enable further analysis of the application. This analysis can be either static, by looking only at the application's source code, or dynamic, by instrumenting the application to collect specific metrics during the execution. An example using static analysis is the Clava auto-parallelization library, AutoPar-Clava [35, 36]. This library analyzes loops and finds dependencies between iterations in order to understand if parallelization is possible and how to apply it via OpenMP.

We use three main ways of transforming the application source code. First, code can be inserted into the application by providing the code to be inserted in a LARA aspect. Code insertions are very flexible and useful for low-level, fine-grained tasks.

Then, Clava actions can be applied, which are transformations applied by Clava on a join point selected by the user. These actions provide an abstraction as the user does not have to control how the transformation is carried out. Examples of such actions include *Loop Tiling*, applied to loops, and *Function Cloning*, which clones the selected function and changes its name to one specified by the user.

Finally, code transformations can be provided by Clava libraries, which can be imported and used in LARA aspects. These libraries provide high-level code transformations for more coarse-grained tasks. For instance, the `Timer` library is used to measure and report time around a provided join point. It manages all implementation details, from including header files to declaring variables to hold temporary values and reporting the execution time. A couple of lines of LARA code can achieve this (as shown in Fig. 9). The implementations of these libraries use the previously mentioned code insertions and actions as building blocks, but are hidden from the user.

Clava offers possibilities to transform the target application at several levels of abstraction, meaning that end users can write their custom and targeted transformation aspects to change their applications in a precise way. On the other hand, it is also possible to write aspects that can be reused on multiple applications, reducing the amount of work for repetitive tasks.

We rely on a source-to-source approach due to the following advantages compared to lower-level representations. First, working at the source code level brings a level of flexibility and portability that is not available otherwise. For instance, after performing transformations, any specific target compiler can be chosen, giving more freedom to the programmers and allowing Pegasus to be used in more cases. Concerning flexibility, a source-to-source approach allows the use of other analysis and transformation frameworks that inspect source code, and it also allows developers to further modify the application source code.

Second, there is possibly a lower entry barrier and a smoother learning curve for anyone using such an approach since the strategies are specified at the same familiar level, using a similar programming specification that developers already use when programming. Lower-level representations would require that users learn and reason using a new model. With Pegasus, end users are both able to program their analysis and transformation strategies and to use the ones provided. A lower-level representation would limit the customization by users.

Third, certain information, such as code structure and naming information, is typically lost when converting source code into lower-level representations. For example, struct field names would be lost, and the user would not be able to specify any analysis or transformation based on those names.

3.4 Synthesis and Integration of the Autotuner

The integration of the mARGOt autotuner [16] and deployment of the target application with a runtime adaptivity layer is one of the fundamental steps in the Pegasus approach.

Some characteristics of the application may not be easily gathered statically and may require dynamic profiling. For instance, features that are directly related to the input are not statically predictable. These include input sizes and sparsity, which can make particular algorithms unfeasible, and memory access patterns that directly depend on the input, and that prevent parallelization and the application of some loop transformations.

However, it is also possible that even dynamic profiling cannot be efficiently used since the running conditions may change during execution. In such cases, an autotuner is required to provide runtime adaptation to changes in the execution context. In Pegasus, Clava libraries support the integration of the mARGOt autotuner into a target application. These libraries provide support to the user in three different phases: configuration, generation of the initial knowledge base, and code insertion for the mARGOt interface.

First, the libraries configure how the autotuner interacts with the application, which includes defining knobs, metrics, and the optimization function that guides the choice of the following settings. In the end, Clava generates the configuration file needed by mARGOt.

Then, the libraries can be used to generate the initial knowledge base. Although mARGOt has an online mode, in which it can learn the application's operating points as it executes, it can also start with offline generated knowledge. We can use the Clava libraries to explore the parameters, i.e., the knobs, and data features, and measure the metrics of interest, e.g., execution time and energy consumption. At the end of the exploration, the library generates an operating points list, which is then used by mARGOt.

Finally, we include a library to ease the insertion of code that interfaces with the actual autotuner code. A LARA strategy selects the points in the code where the knobs should be updated, and then, a function of the mARGOt integration library inserts the needed code, taking into account the previous configuration. It also takes care of other details such as inserting include directives and mARGOt initialization code, reducing the amount of manual work the user needs to perform.

4 Strategies for Software Improvement

Given that the target problem for a performance engineer in HPC is composed of profiling, code optimization, and autotuning, this section presents examples of recurrent use cases and how developers can solve them with Pegasus. We selected strategies covering the steps identified in Section 3 to demonstrate some of the capabilities of our approach.

In particular, Section 4.1 presents the strategy *Time and Energy Measurement* related to **Analysis and Profiling**, Section 4.2 and Section 4.3, respectively, present the strategies *Multiversioning* and *Code Transformations* related to **Strategy Selection and Development**, and, finally, Section 4.4 presents the strategy *Autotuning* related to **Autotuner Optimization** and **Autotuner Integration**.

4.1 Time and Energy Measurement

Fig. 9 shows a simple aspect that instruments arbitrary function calls to measure either the execution time or the consumed energy. We parameterize the presented aspect with the name of the function whose calls we want to measure, and whether to measure energy or time. It uses two libraries that are part of the LARA API, `Timer` and `Energy`.

```
1 import lara.code.Timer;
2 import lara.code.Energy;
3
4 aspectdef MeasureTimeOrEnergy
5     input funcCallName, measureEnergy end
6
7     select call end
8     apply
9         if(measureEnergy) {
10            new Energy().measure($call);
11        } else {
12            new Timer().time($call);
13        }
14    end
15    condition $call.name == funcCallName end
16 end
```

Fig. 9. LARA aspect to advise execution time and energy consumption measurements around a given function call.

Line 7 of the example selects every function call of the input application. The aspect filters these calls with the condition in line 15, i.e., it only transforms calls to functions with names matching the provided name (parameter `funcCallName`). In the `apply` block it is created an instance of the correct library, either `Timer` or `Energy`, and it is passed the call join point (`$call`) to the corresponding function, which surrounds the call site with the code needed to measure the execution time or the energy consumed.

If we weave twice this aspect into the application, first to measure energy consumption and then to measure execution time on the same function call, the resulting application looks like the matrix multiplication call presented in Fig. 3. The original call was instrumented to collect metrics of interest during the execution of the function and to print the metric values to the standard output. The `Timer` and `Energy` libraries also manage the insertion of include directives automatically.

We note that the code of this aspect can be easily extended to consider other types of join points, e.g., loops, code sections, functions with specific characteristics.

4.2 Multiversioning

A recurring transformation performed with Clava is the generation of multiple versions of a target function. We usually follow this transformation by replacing some (or all) of the target function calls with a mechanism that can choose different versions at runtime. Each version can then be optimized separately, and the choice of which to execute is postponed to runtime. Fig. 10 shows a fragment of a simplified version of such a strategy (used in Gadioli et al. [37]), which optimizes each version differently by choosing different compilation flags. In other instances, we also change the code of each version, e.g., through the application of different loop transformations.

We parameterize this aspect with a list of optimization flags and a target function, previously selected by the user. Line 8 creates an instance of `MultiVersionPointers`, a library developed to help with the generation of the control code. It makes an array with pointers to functions with the same signature as the original. Each one of the positions holds a pointer to one of the new versions, and the user provides the mapping (index to function name). At runtime, an heuristic or autotuner can choose what function to use by changing the index. From line 10 to line 24, the strategy iterates through all optimization flags and makes a clone for each one of them, giving the clone a new name based on the original and the flag index. Line 20 takes the newly generated clone and surrounds it with pragmas that instruct the compiler on how to optimize the function. Then, line 23 maps the name of the clone to its corresponding index. The remainder of the aspect has three main parts. First, it globally declares the variables to hold the index and the array of function pointers (lines 26–33). The index variable is the knob that can be controlled by an autotuner. Then, it initializes the array in the main function, which is where the mapping is generated by assigning a pointer to each of the versions to its corresponding position (lines 36–39). Finally, it replaces every call to the original target function with a call to an associated function, pointed to by the corresponding array position. For instance, the call:

```
int result = original_target(first_arg, second_arg);
```

is modified to:

```
int result = pointer_array[index](first_arg, second_arg);
```

This kind of strategy can be extended with other variables to create more complex applications with more potential for performance optimization. For instance, in Gadioli

```
import clava.ClavaJoinPoints;
import antarex.multi.MultiVersionPointers;

aspectdef MultiVersioning
  input opts, $target end

  var globalNameOpt = "multi_version_opts";
  var mvp = new MultiVersionPointers($target, [opts.length]);

  for(var optId in opts) {

    // build the new for each clone
    var opt = opts[optId];
    var newName = $target.name + ’_opt’ + optId;

    // generate clone
    var $clone = $target.exec clone(newName);

    // insert opt pragmas around the clone
    call InsertPragmasAroundClone($clone, opt);

    // add to multiversion controller
    mvp.add(newName, optId);
  }

  var intType = ClavaJoinPoints.builtinType("int");
  select file end
  apply
    // insert global for knob
    exec addGlobal(globalNameOpt, intType, "0");
    // insert global for multiversion controller
    mvp.declare($file);
  end

  // initialize the multiversion controller
  select function{’main’} end
  apply
    mvp.init($function);
  end

  // replace all calls to the target function with
  // the multiversion controller
  for (var $call of $target.calls) {
    mvp.replaceCall($call, [globalNameOpt]);
  }
end
```

Fig. 10. Excerpt of a LARA aspect to generate multiple versions of a target function.

et al. [37], we targeted kernels with OpenMP pragmas, and we added another dimension to multiversioning by also considering two possible values for the proc_bind clause. In the end, we exposed three knobs: the number of threads, compiler optimization flags, and the proc_bind value. These knobs can be controlled manually from the command line or automatically from within the program, e.g., with a user-defined heuristic or even an autotuner.

The decision to use function pointers to deal with multiversioning in this example is merely an implementation choice. Although in this case we used the MultiVersionPointers library to help with the code generation, we provide another library to generate a switch statement to choose the version to call. This switch implementation is better suited when additional layers of indirection are present, e.g., in C++ class methods and templates.

4.3 Code Transformations

Fig. 11 presents an example of a LARA aspect capable of applying Loop Tiling [38] to a selected loop nest. Most of the work is performed by the Clava action tile (line 25), which takes the name of the variable holding the block size (tileVar) and a reference loop ($topLevelLoop) marking where to insert the newly generated loop.

```
 1 import clava.ClavaJoinPoints;
 2
 3 aspectdef LoopTiling
 4
 5   input
 6     $topLevelLoop,
 7     tileVars = {}// Maps control vars to tile variable names
 8   end
 9
10   // Get function body
11   $fBody = $topLevelLoop.ancestor(’function’).body;
12
13   // Int type for tile variables
14   var $intType = ClavaJoinPoints.builtinType(’int’);
15
16   for(var $loop of $topLevelLoop.descendantsAndSelf(’loop’)) {
17     var tileVar = tileVars[$loop.controlVar];
18     if(tileVar === undefined) {
19       continue;
20     }
21
22     // Create tile variable
23     $fBody.exec addLocal(tileVar, $intType, ’64’);
24
25     $loop.exec tile(tileVar, $topLevelLoop);
26   }
27 end
```

Fig. 11. Example of a LARA aspect to perform loop tiling on a loop nest.

We parameterized the presented aspect with the reference loop (which is the outermost loop of the nest), and a map containing the loops to tile. The map, tileVars, maps the names of the control variable of each target loop to the name of the corresponding variable that holds the block size. In this aspect, these variables are declared as integers (line 23) on the scope where the reference loop is located (line 11). Finally, the aspect applies loop tiling to each loop in the map (line 25).

This aspect assumes the loops are on the same loop nest (the tile action fails if they are not) and only requests the user to select and provide the reference loop (e.g., the outermost) and define which loops to tile, identifying them by their control variable inside the loop nest. This aspect is reusable, and we may apply it to multiple loop nests in different applications.

The current version of the Clava compiler supports several built-in code transformations, such as loop tiling (used in the example above) and interchange, function inlining, cloning and wrapping, variable renaming, and setting loop parameters such as induction variable initial value, step and stopping condition. Other code transformations are provided or can be programmed using LARA code and may use built-in code transformations as building blocks.

4.4 Autotuning

This strategy shows how to integrate mARGOt [16] in the target application. The autotuner enhances the original application to deal with changes in the execution context. We assume that the choice of the block size (for instance, from the previous loop tiling transformation) should take into account both the underlying architecture and the size of the input matrices. By augmenting the application with a runtime autotuner, we can make it resilient to changes in the sizes of the matrices, leaving mARGOt to automatically choose the optimal block sizes (or as close as possible to optimal, based on the performed exploration).

Starting from an application with tiled loops (e.g., after weaving the aspect presented in Fig. 11), we can use a Clava library to integrate mARGOt in the application and generate the configuration files. We organized this integration strategy in three steps, which are all called from a top-level aspect: configuration, design-space exploration (DSE), and code generation.

```
 1 aspectdef XmlConfig
 2   input configPath, $targetFunc end
 3   output dseInfo, codeGenInfo end
 4
 5   /* ... */
 6
 7   /* knobs */
 8   matmul.addKnob(’block_size_1’, ’BS1’, ’int’);
 9   matmul.addKnob(’block_size_2’, ’BS2’, ’int’);
10
11   /* data features */
12   matmul.addDataFeature(’N’, ’int’);
13   matmul.addDataFeature(’M’, ’int’);
14   matmul.addDataFeature(’K’, ’int’);
15
16   /* ... */
17
18   /* generate the configuration file */
19   config.build(configPath);
20
21   /* generate the information needed for DSE and code gen*/
22   dseInfo = MargotDseInfo.fromConfig(config, funcName);
23   codeGenInfo = MargotCodeGen.fromConfig(config, funcName);
24 end
```

Fig. 12. Excerpt of a LARA aspect to configure the mARGOt autotuner.

Fig. 12 presents part of the aspect responsible for the configuration step of the overall autotuner integration strategy. For brevity, we omitted some of the code lines. The Clava library allows the users to instantiate a configuration object and then add and configure multiple mARGOt blocks. In this example, we configure a single block named `matmul`. Lines 8–9 and 12–14 show the most important parts of the configuration, where the user can specify knobs, and where the user can specify data features, respectively. Software knobs are what the autotuner controls, and they are modified in response to runtime contextual information changes. In this case, a change is represented by the data features, which are the sizes of the input matrices. The call to the `build` function (line 19) generates an XML configuration file needed by mARGOt. However, the configuration information is not only used to generate this file. The ensuing steps reuse some of this information, which is why that information is propagated forward (lines 22–23).

Fig. 13 shows an excerpt from an aspect that performs DSE and builds the knowledge base used by mARGOt. This aspect evaluates several combinations of values for the knobs (representing the autotuner choices) and values for the data features (simulating changing matrix sizes). The aspect defines the values to test in lines 17–18. From the top-level aspect, this aspect receives a target function and corresponding function call. We select the body of the function and instruct the DSE library to perform the changes in values inside that scope (lines 5–8). We use the call to the target function as the measuring point (line 9), which in this example is only measuring execution time (line 14). After providing this information and how many runs to perform (at the end, this aspect reports the average of 30 runs, as defined in line 11), the code variants are generated, and the data collection begins. The results of the exploration are processed, and the library generates the knowledge base in the format required by mARGOt.

```
 1 aspectdef Dse
 2   input dseInfo, opListPath, $targetCall, $targetFunc end
 3
 4   // Select portion of code that we will explore
 5   select $targetFunc.body end
 6   apply
 7     dseInfo.setScope($body);
 8   end
 9   dseInfo.setMeasure($targetCall);
10
11   dseInfo.setDseRuns(30);
12
13   // add desired metrics
14   dseInfo.addTimeMetric(’exec_time_ms’, TimeUnit.micro());
15
16   // set the knob values
17   dseInfo.setKnobValues(’block_size_1’, 16, 32, 64, 128);
18   dseInfo.setKnobValues(’block_size_2’, 16, 32, 64, 128);
19
20   // set the feature values
21   dseInfo.setFeatureSetValues([’N’, ’M’, ’K’],
22         [32, 16], [16, 16], [64, 64]);
23
24   dseInfo.execute(opListPath);
25 end
```

Fig. 13. Excerpt of an example LARA aspect to perform design-space exploration for the mARGOt autotuner.

Finally, the last step is the generation of the code to interface with mARGOt. Another part of the Clava mARGOt library performs this generation, and Fig. 14 shows an example of its use. In this example, we generate and insert the code to perform an update call to mARGOt right before the selected join point. The aspect selects the loop inside the target function with a control variable matching the one provided as input. This call to mARGOt’s `update` takes the values of the data features and sets the values of the knobs accordingly. Information such as the name of the autotuner block, the names of the variables holding the knob values, and data feature values are all already defined in the `codeGenInfo` object, passed from the top-level aspect. This information was previously defined in the configuration step (line 23 in Fig. 12).

```
aspectdef CodeGen
  input codeGenInfo, $targetFunc, controlVar end

  select $targetFunc.loop end
  apply
    codeGenInfo.update($loop);
  end
  condition $loop.controlVar == controlVar end
end
```

Fig. 14. LARA aspect example to instrument an application with calls to the mARGOt interface.

5 Case Study

The case study is a prototype of a futuristic navigation system, NavSys, being developed in the context of smart cities.

NavSys is a highly sophisticated application, representative of a future generation of navigation systems in the context of smart cities and the management of autonomous vehicles. This application includes components based on methods and algorithms widely used in other domains, such as the identification of shortest paths, betweenness centrality, and Monte Carlo simulations.

Fig. 15 shows a block diagram of the NavSys application consisting of four main components: K-Alternative Paths Plateau (KAP), Probabilistic Time-Dependent Routing (PTDR), Betweenness Centrality (BC), and Routing Reordering and Best Solution Choice (RBSC). KAP is responsible for providing K path alternatives for routing a vehicle from origin to destination. PTDR incorporates speed probability distribution to the computation of the route planning in-car navigation systems to guarantee more accurate and precise responses [39]. BC provides information about centrality nodes in the routing map (a graph) needed to identify critical nodes. RBSC reorders the K alternative paths based on different cost functions (depending on the kind of service requested by the users of the navigation system).

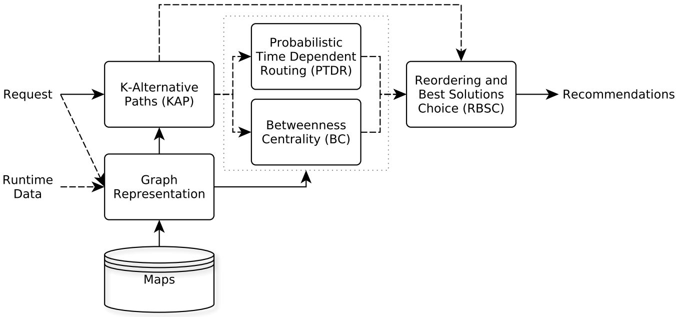

Request
K-Alternative Paths (KAP)
Probabilistic Time Dependent Routing (PTDR)
Betweenness Centrality (BC)
Reordering and Best Solutions Choice (RBSC)
Recommendations
Runtime Data
Graph Representation
Maps

Fig. 15. The structure of the NavSys application.

As NavSys is a computing- and data-intensive application, optimizations are required to reduce the execution time and energy consumption. To provide specific optimizations and an improved version of the application code, we have used the Pegasus approach described in this paper. In particular, we used the Pegasus approach on three components (PTDR, BC, and RBSC), excluding KAP from the analysis. Although the components we optimized are from the same target application, they are independent, and thus they can be seen as different applications from the perspective of our approach.

The NavSys code version used in this paper has been developed by the Czech supercomputing center IT4I to provide an experimental testbed for extending the existing Sygic navigation by server-side routing with a traffic flow calculation for global optimization of city transportation. The NavSys application is a result of recent research on its main components, such as path reordering [40], k-alternative paths [41, 42], betweenness centrality [43, 44, 45], and probabilistic time-dependent routing [39]. Although the complete NavSys application is not publicly available, two of the important components codes, PTDR [39] and BC [43, 44], have been disclosed and are available online[5].

We note that performance improvements for similar components to the ones used in NavSys have been addressed by using hardware accelerators such as GPUs and FPGAs. Examples are the use of GPUs for BC [45] and the use of FPGAs and GPUs for Quasi-Monte Carlo Financial Simulations [46]. Although in this paper we do not target hardware accelerators, it is in our plans to extend the Pegasus approach with strategies for heterogeneous architectures with hardware accelerators. At the moment, the support provided can help developers and performance engineers to identify possible bottlenecks, hotspots, communication patterns via code instrumentation and acquire certain computing and code characteristics (via profiling and static analysis) that can guide decisions regarding offloading to specific components of the target architecture.

5.1 Pegasus Approach in the Case Study

Table 2 presents the classification of each strategy applied to the use case regarding their steps (as described in Section 3) and reusability. The following sections detail the strategies presented here.

TABLE 2
Classification of each strategy applied to the use case.

Component	Strategy	Steps	Reusable
PTDR	Exploration	①, ③	No
	Autotuner Integration	③, ④, ⑤	No
BC	Analysis	①	No
	Production	②, ⑤	No
	EvalDistances	①, ②	No
	EvalMeasures	①, ②	Yes
RBSC	Versioning	①, ②, ⑤	Yes

Out of the seven strategies applied to the use case, we classify five as analysis strategies. The analysis is an essential part of the methodology since it provides the initial knowledge of the application and uncovers details for custom transformations. This information drives and steers the next steps.

The Pegasus approach supports the sequence and progression of the steps in the methodology. We may use analysis and exploration strategies as standalone tools that provide information, or we may use them to guide optimization changes and generate a final production version. For instance, in the BS component, the initial analysis strategy leads to the production strategy that changes the main loop of the application to skip BC computations based on the similarity of the input graphs.

We classify two strategies as performing three methodology steps. First, *Autotuner Integration*, applied to the PTDR component, explores the application design space to build an autotuner knowledge base, integrates the mARGOt autotuner into the application with all the needed configuration, and builds a production application that is ready to be used.

[5] The BC code is available at https://github.com/It4innovations/Betweenness. The PTDR code is available at https://github.com/It4innovations/PTDR

```
$result \leftarrow$ MCSimulation($samples, period$)
$stats \leftarrow$ makeStats($result$)
printStats($stats$)
writeResults($result$)
```

Fig. 16. The original PTDR main task.

Then, the *Versioning* strategy, applied to the RBSC component, changes the application to allow multiversioning, which is used both in analysis and production scenarios.

This work does not explore some possibilities, such as the integration of the autotuner into BC and RBSC. In BC, the autotuner can control the threshold to skip more computations and decrease the execution time and energy consumption, while maintaining the error below a predefined value. In RBSC, the autotuner is used to choose which of the multiple generated versions would run at any given time, taking into account the accuracy of the generated routes and the time taken to compute them.

Finally, Table 2 shows that two of the eight used strategies are reusable, i.e., we could apply them directly to other applications. The *EvalMeasures* strategy uses an aspect that is parameterized with a loop, around which it inserts code to measure both execution time and energy consumption. Such a strategy can be used by other applications to measure other loops (or any other points in the code), by selecting them according to their needs and filters and passing them to this aspect. The *Versioning* strategy, applied to RBSC, is reusable since we parameterized it on several levels, mainly on what reordering functions to evaluate, and what mappings to apply to each input of the reordering functions. In order to be applied in the target code, it only needs a call to a function that we replace with the new versions to test.

5.2 PTDR Exploration

The application component used here, Probabilistic Time-Dependent Routing or *PTDR*, incorporates speed probability distribution to the computation of route planning [47]. Fig. 16 presents the pseudocode of such an application, which performs a Monte Carlo simulation, parameterized with the number of samples. Varying the number of samples introduces a trade-off between faster execution and more accurate results, i.e., a smaller number of samples produces less accurate results, but they are computed faster. Depending on the server load or urgency of the routing request, it is possible to favor one or the other to achieve the goals of the current execution policy. Furthermore, the simulation is parallelized with OpenMP, which allows for the exploration of the number of threads to use and thus more exploitable trade-offs.

We assume that running conditions, such as the server load, may change during execution, which may impact the performance of the application and render the decisions based on the offline exploration unfit for dealing with the current conditions. For this reason, we developed another strategy to integrate mARGOt [16] into the application, in order to provide runtime adaptability capabilities. The goal is to dynamically reduce the number of Monte Carlo samples based on an unpredictability feature, which we extract from a previous (smaller) execution with the current data. The knowledge base needed by mARGOt is provided by the previously described exploration step, while the Clava mARGOt library provides the configuration files and API integration.

In order to perform the PTDR parameter exploration and autotuner integration, we developed two strategies consisting of several LARA aspects. We first analyze the application in order to understand how to properly configure it and then add the autotuner to improve the selected parameters under dynamic runtime conditions.

5.2.1 Exploration

The first strategy, *Exploration*, apply Design Space Exploration (DSE) to the original application. To perform DSE, we use a LARA library, which allows us to define how to compile and run an application, and which code variables to change and how. It is also possible to measure execution time, energy, and other user-defined metrics. This LARA library receives as parameter the number of executions to perform per variant, starts the exploration process, and returns, for each metric, the average of the collected values of all executions.

In this exploration, we want to measure the impact of the number of samples and threads. To achieve this, we specified, in the LARA strategy, which variables are changed and tested. This change is performed inside a user-defined scope, which in this case, is a block (or compound statement) surrounding the simulation call. The values tested for the number of samples and threads are {500, 1000, 5000, 10000, 50000, 100000, 500000, 1000000} and {1, 2, 4, 8, 16, 32, 64}, respectively. The LARA library automatically generates all code versions for the 56 (8×7) variants, compiles, and runs each of them.

For each version tested, the LARA code collects the metrics defined by the user. It is also possible to specify the scope where these metrics are collected. In this case, we collected execution time and energy consumed around the call to the Monte Carlo simulation. We developed a custom error metric, specific to PTDR, called `PtdrErrorMetric` to study the effect of reducing the number of samples on the accuracy of the results. This user-defined metric can be instantiated and provided to the LARA library, so it is possible to measure it alongside the time metric. To define a new metric, we need to extend the base metric class and implement two methods, one that instruments the application and one that extracts and reports the metric value.

The value of the error is the Mean Squared Error of the obtained results for the percentiles {5, 10, 25, 50, 75, 90, 95} in comparison to a reference value, which we obtained by simulating with 1,000,000 samples and the maximum number of threads in the machine.

This strategy can be parameterized to also extract another metric from the running application, which is the *unpredictability* value, as calculated by the application's statistical report. The process of extracting this metric involves a slight transformation of the source code, specified in the `ExposeUnpredictability` aspect, which is conditionally called by the `Exploration` aspect. Before the original call to the Monte Carlo simulation, the strategy inserts a clone of that call with a minimal number of samples. Then, we

```
$testResult \leftarrow$ MCSIMULATION($testSamples, period$)
$testStats \leftarrow$ MAKESTATS($testResult$)
$unpred \leftarrow testStats.variationCoeff$
$samples \leftarrow$ MARGOTUPDATE($samples, unpred$)
$result \leftarrow$ MCSIMULATION($samples, period$)
$stats \leftarrow$ MAKESTATS($result$)
PRINTSTATS($stats$)
WRITERESULTS($result$)
```

Fig. 17. The original PTDR main task after being woven with the autotuner strategy.

use the statistical report of the application to extract the variance of the obtained results for that particular input. The code inserted by the LARA strategy collects this information with a `VariableValueMetric`, which prints the value of a user-defined variable in the measurement scope.

5.2.2 Autotuner Integration

The second strategy, *Autotuner Integration*, enhances the application with autotuning capabilities (via the mARGOt autotuner) by performing three main tasks. First, the mARGOt LARA library is used to configure how the mARGOt autotuner interacts with the application. We specify a set of configurations about the operation of the autotuner, including the definition of knobs, metrics to collect, and the optimization functions to use. With this information, the library produces an XML file that otherwise would have to be specified manually.

Secondly, we use the previous exploration strategy to perform a new DSE targeted to the integration of the autotuner. This time we decide not to explore the number of threads and extract the *unpredictability* metric, which is used by mARGOt as a *data feature*. A data feature is input data that the autotuner takes into account for the update of the knobs. After the exploration finishes, the library converts the DSE results into the XML file that mARGOt uses as the initial knowledge base for the autotuning process.

Finally, this library is used again to insert code in the application to call the mARGOt API. The strategy selects the point of interest, the call to the Monte Carlo simulation, and defines it as the update point, where mARGOt is called to update the knob controlling the number of samples. The LARA library automatically takes care of the implementation details, such as the generation of the code to be introduced. This step uses information previously defined in the first step, e.g., the knobs and data features, to generate the correct code without relying on the user providing the same information twice. Fig. 17 shows the pseudocode of the resulting application. The first step is to call the simulation with a minimal number of samples to extract the unpredictability of the input data. Then, we pass this information to the autotuner so it can choose the best-suited number of samples to use.

5.3 BC Exploration

Let us consider an application that periodically computes the Betweenness Centrality (BC) [43] over instances of graphs representing routing maps of cities and traffic information. This computation is expensive and for a large city or using a very detailed graph, it may require a long time to complete.

```
for every graph update do                ▷ can also be periodic
    $graph \leftarrow$ LOADGRAPH()
    $result \leftarrow$ BETWEENNESS($graph$)
end for
```

Fig. 18. The original BC main task.

Fig. 18 presents the pseudocode showing the main BC task. Every graph is loaded and used immediately to calculate the BC of its nodes. In this case, we consider that changes in the graph sent to a file communicate traffic flow information and route state (other possible optimizations may consider in-memory graphs).

We explored the idea of skipping some computations of BC and approximate them with previously computed BC results. We would only perform this approximation if the inputs of the computation, the graphs representing routing maps and traffic, were considered similar. It is important to note that we consider that no new routes are added, and thus graphs that represent routings in cities always have the same structure, and the edge weights are the only possible changes in the graph. Skipping these computations would save execution time and energy since the computation of graph similarity is faster and scales linearly with the number of nodes.

We consider two graphs similar if their D distance is less than a defined threshold value, T. In the experiments, we used a distance defined as

$$D = \frac{1}{E}\sum_{n=1}^{E}\left|W_n - W_n^{'}\right|,$$

where E is the number of edges in the graph, and W_n and $W_n^{'}$ are the weights of the n^{th} edges of the current and previous graphs, respectively.

In order to evaluate how skipping BC computations affects the accuracy of the system that depends on these results, we measure the difference between the reused result and what would be the computed result for the current input. In our case study, the result of a BC computation is a list of nodes (always in the same order) and their corresponding centrality. Our first step is to compute the rank of each node, meaning the node with the highest centrality has rank 1, and the node with the lowest centrality has rank N, where N is the number of nodes in the graph. After this, we compute B, the Euclidean distance of the vectors formed by the ranks of the two results.

We note that the LARA strategies used (and described below) can be easily extended to provide other distance metrics and woven code based on those metrics.

To achieve the reduction in execution time and energy consumption, while also maintaining accurate results, we developed strategies for analysis of the problem, generation of production code, and evaluation of the results.

5.3.1 Analysis

The first strategy, *Analysis*, rewrites the original application to compute D and B in consecutive iterations of the main loop.

```
for every graph update do
    $graph \leftarrow$ LOADGRAPH()
    $D \leftarrow$ CALCDIST($graph, previousGraph$)
    SAVE($D, arrayD$)                          ▷ save consecutive $D$s
    $previousGraph \leftarrow graph$
    $result \leftarrow$ BETWEENNESS($graph$)
    $B \leftarrow$ CALCEUCLIDEAN($result, previousResult$)
    SAVE($B, arrayB$)                          ▷ save consecutive $B$s
    $previousResult \leftarrow result$
end for
PRINT($arrayD$)
PRINT($arrayB$)
```

Fig. 19. The BC main task after being woven with the analysis strategy.

```
for every graph update do
    $graph \leftarrow$ LOADGRAPH()
    $D \leftarrow$ CALCD($graph, previousGraph$)
    if $D < T$ then
        $result \leftarrow previousResult$          ▷ BC skip
    else
        $result \leftarrow$ BETWEENNESS($graph$)
        $previousResult \leftarrow result$
        $previousGraph \leftarrow graph$
    end if
end for
```

Fig. 20. The BC main task after being woven with the production strategy.

Fig. 19 presents the pseudocode of the BC main task after being woven with the Analysis strategy. We calculate the distance D after loading the graph for the current iteration by comparing it to the graph of the previous iteration. Similarly, B is calculated after computing BC for the current iteration and comparing it to the result of the previous iteration. We save and print these distances at the end of the execution of the loop and then use them to suggest the threshold T.

5.3.2 Production

The second strategy, *Production*, prepares the application to use the described approach to skip BC computations. We parameterized it with T, the threshold to use.

Fig. 20 shows the pseudocode of the central BC task resulting from weaving the original application with this production strategy. The strategy inserted the computation of the graph distance D and a mechanism to reuse the previous result if this distance is less than the predefined threshold. If it is not, BC is computed for the current input and made available for reuse by saving both the current graph and the current BC result.

5.3.3 Evaluation

We developed two strategies for the evaluation and tuning of the production application. The first, *EvalDistances*, changes the application in order to collect statistics of the B distance based on the production version of the application. This strategy generates a version that is similar to the production version but always computes BC. Then, whenever it was supposed to be a skip ($D < T$), it records the distance between the current BC result and the one that would be used in case there was a skip. At the end of execution, it reports how many skips happened and the minimum, maximum, and average B for those iterations.

```
for $route \in routes$ do
    $score \leftarrow$ EVALUATE($route$)
    SAVE(score, scores)
end for
```

Fig. 21. The original RBSC main code.

The second evaluation strategy, *EvalMeasures*, is concerned with measuring the execution time of the application, considering three versions, original, parallel, and production. This is a straightforward strategy that selects the main loop of the application and surrounds it with calls to a library to measure time using standard C++ libraries. The EvalMeasures strategy can be parameterized to call the Production strategy before inserting the measurement code. This way, it is possible to measure both the original and the production versions.

5.4 RBSC Exploration

After the generation of the several possible paths for a NavSys request, they can be evaluated and reordered, in order to reduce the total driving time. The pseudocode of this application is presented in Fig. 21. However, there is no single optimal reordering, as it is dependent on the current data and requirements (e.g., customers may require a NavSys service with more reliable data and accuracy by using BC and PTDR). In order to satisfy these different scenarios, we explore several evaluation heuristics that can provide acceptable solutions to the reordering problem. We developed a LARA strategy that can be used to create and explore several heuristics automatically.

5.4.1 LARA Strategy

The strategy assumes that this application component calls a reordering function. The strategy replaces the call to the evaluation function with a `switch` statement, controlled by a user-defined expression, which calls one of the heuristics generated by the strategy (the expression can be any valid C/C++ expression, such as a macro, or a variable within scope). Other kinds of strategies, automated with LARA, could generate a specific version of NavSys for each customer requirement. The pseudocode of the application after weaving is presented in Fig. 22. The transformed application can then be used to automatically explore the different heuristics by selecting different values for the `switch` statement.

Fig. 23 presents the pseudocode of the LARA strategy that modifies the code to enable exploration of evaluation heuristics. It receives several inputs, some of them optional:

- **rcall**: the evaluation function already present in the target application, which will be replaced by the `switch` statement.
- **extraValues**: maps names of variables to functions that are available in the scope of the object where

```
for $route \in routes$ do
    switch $version$ do
        case $0$
            $score \leftarrow$ EVALUATE($route$)
        case $1$
            $score \leftarrow$ HEURISTIC1($route$)
        ...
        case $N$
            $score \leftarrow$ HEURISTICN($route$)
    end switch
    SAVE(score, scores)
end for
```

Fig. 22. The RBSC code after weaving.

the original evaluation function (`rcall`) resides (e.g., instance functions, static functions). Before each version of the reordering function is called, the functions defined here are called and stored in a variable with the given name. The *mappings* input can then use this variable. This input is optional.

- **mappings**: array where each element is a C/C++ expression that maps variables of the current object instance to values used as inputs to the reordering functions.
- **type**: the output type of the mappings. If none is specified, assumes the type is `double`.
- **functions**: array with names of functions that are available in the scope of the object where the original reordering function resides. Each reordering function must have some inputs of the same type as `type`, and the number of inputs can range from 0 to N, where N is the length of the array `mappings`.
- **switchCondition**: the expression that is the condition of the generated `switch` statement. The `switch` assigns an incremental integer value to each version of the reordering functions, starting from 0. The value 0 is always associated with calling the original function.

In the first step of the algorithm, the definition of the reordering function called in the original application is obtained. If a definition is not found, e.g., the code of the function is not available, the algorithm cannot continue.

Next, if `extraValues` is not empty, it creates the variable declarations that contain the value returned by calling the corresponding function. These variables are stored in `decls` and are available for the mappings.

After this, the mapping functions are created and stored in the array `mappers`, which is then used to create the several reordering implementations stored in the array `reorders`. The reordering implementations are generated based on the given mappings and reordering functions. For each reordering function, there are as many reordering implementations as combinations of mappings for the function inputs. Currently, the reordering implementations are generated using combinations, but it is possible to change the strategy to use permutations instead.

```
Input
    $rcall \leftarrow$ the reordering call already in the application code
    $extraValues \leftarrow$ maps new variable names to already existing functions.
    $mappings \leftarrow$ array of available mappings to explore
    $type \leftarrow$ the return type of the mapping functions
    $funcs \leftarrow$ array of reordering functions to explore
    $switchCondition \leftarrow$ expression that controls the switch statement

$def \leftarrow rcall.definition$
$decls \leftarrow$ EXTRAVALUESVARDECLS($def, extraValues$)
$mappers \leftarrow$ MAPPERS($mappings, def, type, decls$)
$reorders \leftarrow$ REORDERS($def, decls, funcs, mappers$)
CREATESWITCH($rcall, switchCondition, reorders$)
```

Fig. 23. The strategy that modifies the code to enable exploration of reordering functions.

Finally, the original reordering call (i.e., `rcall`) is replaced by a `switch` statement, which uses the `switchCondition` as its condition expression, and for each case it calls a reordering implementation, except for case 0 that executes the original reordering call.

6 Evaluation

This section presents the use of Pegasus in the performance engineering process targeting the components of the NavSys application: Probabilistic Time-Dependent Routing (PTDR), Betweenness Centrality (BC), and Routing Reordering and Best Solution Choice (RBSC).

Table 3 presents the main characteristics of the machine and environment used to perform the experimental evaluations. The hardware in the machine used in this evaluation is configured to represent a single node of an HPC machine, namely, one of the machines in the IT4Innovations supercomputing center[6].

TABLE 3
Specifications of the machine where the experiments were performed.

Parameter	Value
CPU	2x Intel Xeon CPU E5-2630 v3 @ 2.40GHz
RAM	128GB
OS	Ubuntu 16.04 LTS
Kernel	4.11.0-kfd-compute-rocm-rel-1.6-148
Compiler	GCC 7.3
Flags	`-fopenmp -O3 -march=native -std=c++14`
OpenMP	OpenMP 4.5

The execution time measurements were performed with standard Linux libraries with calls automatically injected to the applications' code using LARA strategies. We measured the amount of energy consumed with our library, SpecsRapl[7], which is a wrapper around RAPL [29]. Calls to our library were also automatically injected into the application.

[6] https://www.it4i.cz/?lang=en

[7] The code for SpecsRapl can be found at https://github.com/specs-feup/specs-c-libs

6.1 PTDR Evaluation

This section presents the results collected during the evaluation experiments. The results have different focuses, from an estimation of the work performed by Clava and the described strategies (comparing to a manual alternative), to the analysis of possible tradeoffs.

We invoked PTDR with speed profiles for the UK. For the *Exploration* strategy, the number of samples and threads were controlled by the exploration parameters, as described previously. For the *Autotuner Integration* strategy, the application always uses 32 threads. We measured the overall execution time of these explorations with standard Linux programs, from the moment the JVM is launched to the moment where it terminates execution.

Table 4 presents statistics about the aspect files developed to implement the strategies previously described. The first five files are all used in the Exploration strategy and called from the `Exploration.lara` file, the strategy entry point. The main code of the autotuner integration strategy is defined in `MargotTuning.lara`, but all other files are used as well since this strategy relies on the previous DSE to build the knowledge base for mARGOt. The first column shows the number of logical lines of source code (SLoC), i.e., a line count disregarding certain elements such as comments and closing brackets. The aspects are relatively short since the definition of the exploration is performed at a high level, and we do not insert a large amount of native code. The outlier is the `MargotTuning.lara` file, with 105 SLoC. However, around half of those lines are pure JavaScript used to translate between different data formats and to generate the XML configuration file. The second column of the table shows the number of aspect definitions (`aspectdef`s) in each file. Aspect definitions are the main modular unit in LARA and allow organizing code into particular concerns which can be reused and parameterized. Similarly, we also organize the native code inserted into the application (present inside the LARA files) into code definitions, or `codedef`s, which are template-like functions for native code. A special note is needed for the files `VariableValueMetric.lara` and `PtdrErrorMetric.lara`, which do not have any aspects. As mentioned before, to implement a custom DSE metric, we need to extend a JavaScript class. These two files, which contain only JavaScript code, are simply implementing metrics.

TABLE 4
Lines of code of the developed LARA aspects split in files.

File	SLoC	Aspects	Comments
Exploration.lara	58	2	9
ExposeUnpredictability.lara	13	1	3
VariableValueMetric.lara	22	0	8
PtdrErrorMetric.lara	28	0	11
InstrumentPtdrErrorMetric.lara	10	1	0
MargotTuning.lara	105	3	42
Total	236	7	73
Average	39.33	1.17	12.17

Table 5 contains weaving metrics for the previously described strategies when woven into the application. For each strategy, we show the number of called aspect definitions, the number of iterated join points (LARA objects that represent code points and can be manipulated by the user), the number of join point attributes queried (e.g., for filtering) and the number of actions taken on the selected join points (and how many of those were code insertions). The metrics show that Clava automatically iterates over a large number of points in the code, according to the user specifications (the LARA `select` blocks). This iteration is performed hierarchically, starting at the file level, then going through all functions and then to the fine-grained points, such as specific statements, function calls, or loops. The results also show that a large number of attributes are queried to find the target points in the code. In the case of these strategies, the attributes are mainly the names of functions and function calls. A user could manually go through these structures in the source code and find where the points of interest are. However, with Clava and LARA, one can automatically iterate over the source code of an extensive application, filter, and capture the points needed.

The last column of Table 5 shows the actions applied to the selected points, i.e., the set of code structures (calls or functions in this example) that we get after filtering based on their attributes and hierarchical structure. Since the strategies described previously rely mainly on adding extra functionality, they can be mostly accomplished with insertions of code into the original application at the target locations. The Exploration strategy tests a broader set of parameters, which results in more generated versions and more performed actions.

TABLE 5
Weaving statistics for the strategies applied to the application.

Strategy	Aspects	JPs	Attributes	Actions (Inserts)
Exploration	847	51557	47736	1456 (840)
Autotuner	133	9040	7830	214 (123)
Total	980	60597	55566	1670 (963)
Average	490	30298.5	27783	835 (481.5)

Table 6 shows the number of lines of code in the original application and the versions resulting from the weaving. The second column shows the difference to the original version of the application. It is important to note that while the Exploration strategy appears to change the application only slightly, it automatically generates 56 different versions, each with different numbers of threads and samples, that are then compiled and executed to collect the metrics. The number 399 for the *Autotuner Integration* strategy is the number of lines in the final application, the production version with support for mARGOt. However, during the exploration, the strategy automatically generates eight files with 410 SLoC each, one per value of the number of samples tested, as described previously. Furthermore, this strategy also generates two XML for mARGOt with 26 (the configuration) and 107 lines (the knowledge base).

The Exploration strategy went through the various values of the number of threads and samples and collected the three metrics (execution time, energy consumed, and error) for each generated and tested version. This exploration took 28080 seconds. The results of this design-space exploration

TABLE 6
Lines of code of the application, originally and after being woven with each aspect.

Version	SLoC	Δ SLoC
Original	394	0
Exploration	410	16
Autotuner Integration	399	5
Total	1203	21
Average	401	7

are presented in Fig. 24. These heat maps have the exploration parameters on each of the axes, and the cells contain the value of each specific metric.

The results show the execution time (in seconds), the energy consumed (in joules), the error of the computation resulting while varying the number of threads and samples, and the average power (in watts). As expected, both the execution time and energy consumed decrease with the number of threads and increases with the number of samples. The error metric behaves more erratically for smaller numbers of samples. However, as the number of samples increases, the behavior becomes more consistent.

As for the mARGOt autotuner integration strategy, we were primarily concerned with its integration using our approach and how it would compare to the alternative of manually integrating it. We think there are definite advantages to using a semi-automated approach like the one presented here. The (automated) design-space exploration to build mARGOt initial knowledge base took 1220 seconds. We could then use this to provide runtime adaptation. Thanks to the usage of the Pegasus approach, we could reduce the number of simulations by a significant amount, between 36% and 81%, with a negligible code overhead.

6.2 BC Evaluation

This section presents the results collected during the evaluation experiments, which range from an estimation of the work performed by Clava and the described strategies (comparing to a manual alternative), to the comparison of the performance of the original and generated versions.

In these experiments, we tested the original version, a parallel version of the original, and the production version. The parallel version derives from the original by using OpenMP directives on the BC kernel, and we obtain the production version by weaving the parallel version with the *Production* strategy.

This application was invoked with an input of 68 graphs, each with 37812 nodes and 85273 edges. Each graph represents the traffic conditions of the city of Vienna at 15 minutes intervals, from 04h00 to 20h45. We collect the time and energy measurements presented around the main loop of the application (as described in Fig. 18), and they take the loading of the graph file into account (however, this is negligible in the overall execution time and energy consumed).

Table 7 presents statistics about the aspect files developed to implement the strategies described previously. The code in the file `MeasureLoop.lara` is used in the evaluation strategy and called from the aspects in the file `EvalMeasures.lara`. We measure time and energy according to the LARA aspects in these files, which are reused for every version tested. The first column of the table presents the number of logical lines of source code in each file. A considerable portion of the lines of code in these files is the native C++ code that is inserted into the application, mainly functions to calculate distances and collect results. The second column presents the number of aspect definitions (`aspectdef`s) in each file, giving an idea of how well distributed the source code is across the main modular unit of LARA, the aspect definition.

TABLE 7
Lines of code of the developed LARA aspects, divided by file.

File	SLoC	Aspects	Comments
Production.lara	46	3	2
Analysis.lara	87	3	9
EvalDistances.lara	95	4	6
EvalMeasures.lara	14	1	0
MeasureLoop.lara	10	1	0
Total	252	12	17
Average	50.4	2.4	3.4

Table 8 contains weaving metrics for the previously described strategies when woven into the application. The results show, for each strategy, the number of called aspect definitions, the number of iterated join points (JPs), the number of queried attributes, and the number of actions taken on the selected join points.

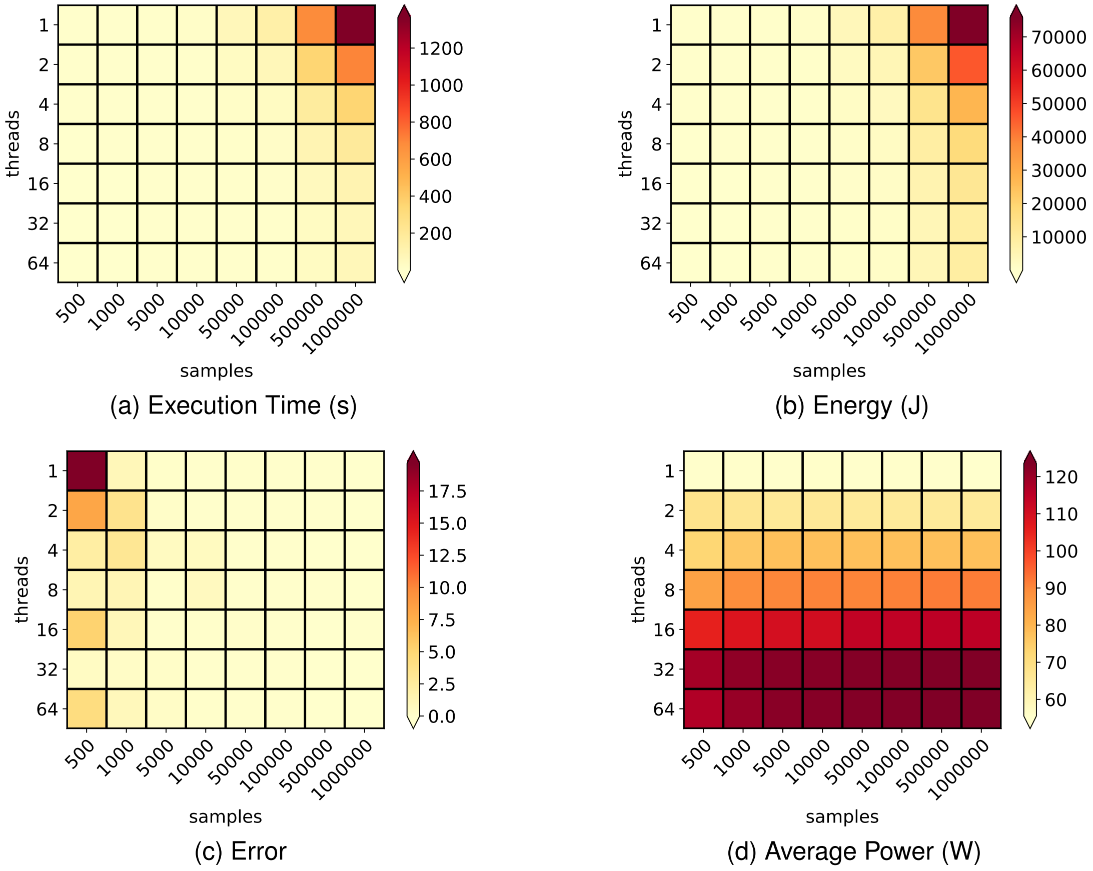
TABLE 8
Weaving statistics for the strategies applied to the application.

Strategy	Aspects	JPs	Attributes	Actions (Inserts)
Production	3	1161	449	5 (5)
Analysis	3	1847	789	11 (11)
EvalDistances	4	1742	686	10 (10)
EvalMeasures	9	1616	833	18 (12)
Total	19	6366	2757	44 (38)
Average	4.75	1591.5	689.25	11 (9.5)

The strategy `EvalMeasures` has more aspect calls and (non-insertion) actions since it is more complicated than the other strategies. The application that results from applying this strategy measures execution time and energy consumption. The additional actions, automatic and transparent to the user, are related to the insertion of header inclusion directives that provide the libraries to collect the data. Similarly, the extra aspect definitions calls are internal aspects for verification and the correct generation of the measuring code. Once again, this is automatic and not seen by the user.

Table 9 shows the number of lines of code in the application, in its original version, and after being modified by each presented LARA strategy. The second column shows the difference in the application size between the original and each generated version. As a note, this metric only counts how many more lines the woven version has, and it does not account for other application-transforming effects such as the replacement of certain statements that the presented strategies use.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2020.3001257, IEEE Transactions on Software Engineering



(a) Execution Time (s)

(b) Energy (J)

(c) Error

(d) Average Power (W)

Fig. 24. Heatmaps with the results of the design-space exploration of PTDR.

TABLE 9
Lines of code of the application, originally and after being woven with each aspect.

Version	SLoC	Δ SLoC
Original	359	0
Production	371	12
Analysis	408	49
EvalDistances	409	50
EvalMeasures	378	19
Total	1925	130
Average	385	26

Based on some empirical testing, we chose a threshold value of $T = 2.00$ since it leads to skipping around a quarter of the total computations (performance measurements are shown later), and the minimum and average distances B are sufficiently close to the values obtained with smaller thresholds.

Table 10 shows the number of BC calls (*BCs*), the execution time (*T*), the energy consumed (*E*) and the average power (*P*) for three versions of the application, the original, the parallel version (*OpenMP*) and the production version with threshold $T = 2.00$ (*Skip*). The results also show the speedup that each generated version achieves when compared to the original (*S*), as well as the improvement in energy consumed (*I*). It is important to note that we generated the production version on top of the parallel version and, when compared to this, it achieved a speedup of 1.32 and an energy consumption improvement of 24.79%. This improvement appears to scale linearly with the number of computations that are skipped. Note that with $T = 2.00$, the application skips 17 out of 68 BC computations or 23.53% of the total, and from the parallel version to the production version, the execution time improves by 23.47%. These results are positive, as it appears both execution time and energy consumption may be reduced linearly with the number of computations skipped. With this knowledge, it is then up to domain specialists to find the optimal threshold that leads to the best performance improvements while keeping acceptable (accurate) BC results, which may change from one application to another.

TABLE 10
Number of calls to BC, total execution time, energy consumed and average power for the original version, the parallel version (*OpenMP*) and the production version (*Skip*) with threshold $T = 2$.

Version	BCs	T (s)	S	E (J)	I	P (W)
Original	68	51198	1.00	2.93×10^6	0.00%	57
OpenMP	68	4934	10.38	5.95×10^5	79.69%	121
Skip	52	3728	13.73	4.48×10^5	84.73%	120

6.3 RBSC Evaluation

This section presents the results collected during the evaluation experiments for RBSC, which estimates the work performed by Clava and the described strategies (comparing to a manual alternative), on four different scenarios based on the strategy described in Section 5. The different scenarios change the number of reordering functions and their arity, as well as the number of mappings to use, which leads to a different number of variants generated. A summary of these scenarios is presented in Table 11.

TABLE 11
The configuration of each tested scenarios.

Scenario	Functions	Arity	Mappings	Variants
Scenario 1	1	2	3	3
Scenario 2	1	3	3	1
Scenario 3	2	2, 3	3	6
Scenario 4	2	2, 3	4	10

Table 12 presents statistics about the aspect files developed to implement the reusable multiversioning strategy that we applied to the RBSC component. There are also other LARA files, one for each scenario, but these only define the specific functions and mappings to use. We do not account for these scenario-defining files. The *SLoC* column shows the number of logical lines of source code, and we can see the aspects are relatively short since they have well-defined concerns. For instance, *Switch* has two aspects used to generate a `switch` statement that allows controlling which of the generated versions is used. The *Aspects* column of the table shows the number of aspect definitions (`aspectdef`s) in each file, and we can further see how well contained each specific concern is. On average, each aspect definition has around 16 SLoC.

TABLE 12
Lines of code of the developed LARA aspects split in files.

File	SLoC	Aspects	Comments
Mappers.lara	32	3	4
Reorders.lara	51	3	10
ExtraValues.lara	30	1	9
Switch.lara	23	2	7
Reordering.lara	22	1	9
Utils.lara	26	1	6
Total	184	11	45
Average	30.7	1.83	7.5

Table 13 contains weaving metrics for the presented scenarios when woven into the application. We show the number of aspects called, the number of iterated join points and their queried attributes and the number of actions taken on the selected join points and how many of those were code insertions. We can see that the strategy automatically iterates over a large number of code elements, some of them in input code, some of them generated during the weaving process, and performs a large number of queries. In this case, process automation is essential, since the number of variants scales exponentially with the number of functions, their arity, and the number of mappings defined by the user.

Table 14 shows the SLoC in the application, in its original version, and after being modified by each scenario. Column Δ *SLoC* of the table shows the difference in the application code size between the original and each of the generated versions. Since the strategy generates new versions according to the reordering of functions and mappings defined by the user, the number of new lines of code scales with the complexity of the scenario.

TABLE 13
Weaving statistics for the strategies applied to the application.

Strategy	Aspects	JPs	Attributes	Actions (Inserts)
Scenario 1	10	6250	527	38 (27)
Scenario 2	8	4474	412	23 (16)
Scenario 3	12	8029	639	47 (34)
Scenario 4	18	15097	1075	101 (75)
Total	48	33850	2653	209 (152)
Average	12	8462.5	663.25	52.25 (38)

TABLE 14
Lines of code of modified files in the application, originally and after being woven with each aspect.

Version	SLoC	Δ SLoC
Original	53	0
Scenario 1	93	40
Scenario 2	74	21
Scenario 3	104	51
Scenario 4	169	116
Total	440	228
Average	110	57

6.4 Threats to Validity

The central claims about the Pegasus approach presented in this paper include some threats of validity, especially in terms of the productivity enhancements to apply the presented performance engineering methodology. In this section, we identify the major threats to validity, explain why we consider them, and discuss extensions to minimize those threats.

The use case considered in the evaluation of Pegasus presented in this paper, despite allowing us to address all the stages of the performance engineering methodology, did not expose the approach to many of its features. We note, however, that previous evaluations in the context of other applications allowed us to conclude about similar effectiveness and efficiency.

The learning curve needed to learn the LARA language was not quantified and might interfere with the adoption of the approach by developers addressing HPC applications. The adoption of JavaScript code for programming part of the strategies might require additional efforts and reluctance by developers with C/C++ background. In this case, a possibility could be to extend the LARA framework and adopt a subset of C/C++ code for LARA strategies.

Other applications may need code transformations that are not outright supported by the current version of the Clava compiler. In this case, future work should extend the portfolio of code transformations.

The efficient use of the autotuner may require the extraction of application-specific metrics that are not present in the original application code. Since these are application-specific metrics, there is not a general library that can be provided with Pegasus to help with their extraction, meaning that specific refactorings are the responsibility of the end user. In this case, the application of the methodology may require the maintenance of different versions of the code and manual efforts.

The evaluation included in this paper does not consider the parallelization of the applications considering multiple computing nodes and, e.g., using MPI. Although we have previously applied code transformations for MPI-based parallelization, our source-to-source compiler does not include strategies for MPI-based auto-parallelization. Besides, we did not evaluate Pegasus when the performance engineering methodology needs to target heterogeneous computing systems, possibly using hardware accelerators, such as GPUs. Although our approach can help developers to generate the host code for communicating and interfacing with OpenCL kernels and may also help to inject code that dynamically selects the offloading of the computations according to runtime data, it is dependent on the existence of the kernels in OpenCL, and further extensions need to integrate compilers able to generate OpenCL code from C/C++.

7 Related Work

To the best of our knowledge, the Pegasus approach presented in this paper is the first one considering the support of the main stages of the performance engineering methodology presented in this paper and followed in the context of HPC applications. We note, however, that there have been research efforts addressing stages of the methodology, and we present and discuss some of the most relevant approaches.

There have been various research efforts focused on providing support for guiding and controlling compilers and toolchains. Recent trends propose the use of DSLs, embedded or external, as a programmable way to allow compiler and toolchain users. However, most of those approaches are focused on subsets of the main tasks needed in the typical methodology described in the introduction of this paper, which would force users to know more than one DSL, e.g., one for instrumenting the code, another one for defining code transformations, and usually impose adoption barriers. The LARA approach has been one of the research efforts contributing to the foundations of a DSL focused on providing abstractions and mechanisms for programming strategies in different levels of development and layers of toolchains. Previous work on LARA has demonstrated its use and usefulness in different contexts and partially uncover its possible contribution when targeting HPC systems.

Because LARA is a DSL addressing strategies for various targets and goals, one can provide higher-levels of abstraction by developing APIs on top of LARA. This usage is exemplified in several cases, mostly dealing with instrumentation and focused on its use with different target languages, or by designing DSLs focused on specific goals and using LARA and its infrastructure as backend. Examples of approaches primarily focused on cross-cutting concerns are the ones typically provided by the AOP communities, for instance, AspectJ [48] and AspectC++ [49]. Orthogonal to those approaches and the main focus of this paper are the ones proposing DSLs to program sections of software applications and supported by code generators empowering the knowledge exposed in more general domain levels than by using general languages. However, the use of those DSLs depends on the application domain and requires adopting different DSLs in the methodology, according to the target domain.

In order to apply code refactoring in early phases of software development and to enable the use of refactoring by inexperienced developers, Liu et al. [50] propose a monitoring-based framework to drive users on applying code refactoring. The refactoring is based on the quality of code in terms of maintainability, extensibility, and reusability. In the context of reducing energy consumption in mobile android apps, authors have also focused on code refactoring. For example, Morales et al. [51] propose a recommendation system for refactoring eight types of anti-patterns.

There are source-to-source compilers that can be used to achieve the kind of transformations performed by the Clava compiler. For instance, Cetus [52] is a source-to-source compiler for ANSI C programs, and ROSE [53] is a source-to-source compiler providing program transformation and analysis tools for C, C++ and Fortran applications. To the best of our knowledge, they cannot easily support all the tasks required in our proposed approach. To apply strategies like the ones presented in this paper, Cetus and ROSE require the implementation of each new strategy internally (with the programming language used to develop each compiler), using lower-level and IR-specific abstractions for each particular compiler, and then to rebuild the compiler including options to apply such strategies. Specifically, they do not easily support end user programming of the required strategies that can vary according to the application, target machine, and requirements, and thus make a built-in integration of some strategies not an option. Albeit possible, this option would require a compiler expert for each compiler and would neither be practical nor reasonable for any end user. On the other hand, while using Clava, users describe the transformations in LARA, which was designed specifically for the analysis and transformation of source code at a high level of abstraction. Additionally, there are other limitations with these compilers that favor the use of Clava. For instance, Cetus only accepts ANSI C. In contrast, since Clava uses Clang to perform the parsing, it accepts a wide range of C and C++ code.

Recognizing the need to apply and specify code transformations, many approaches focus on specific kinds of transformations. For instance, CHiLL [54] is a declarative language focused on recipes for loop transformations. CHiLL recipes are scripts written in separate files, which contain a sequence of transformations to be applied in the code during a compilation step. The PATUS framework [55] defines a DSL specifically geared toward stencil computations and allows programmers to define a compilation strategy for automated parallel code generation using both classic loop-level transformations (e.g., loop unrolling) and architecture-specific extensions (e.g., SSE). Locus [56] is a system and a language for program optimizations that relies on a separation of concerns. Program transformations and optimizations are specified on files separated from the application code and interconnect with it via user annotations that identify sections of code. Besides the programmability for orchestrating compiler optimizations leveraged on interfaces and extensions to existent source-to-source compilers such as ROSE [53] and Pips [57], Locus also provides the interface to optimization-space exploration frameworks such as Opentuner [58].

The Clava compiler relies on the LARA language, which is general enough to be able to describe and implement the approaches and kind of transformations proposed by CHiLL [54] and Locus [56]. With Clava+LARA, one can select elements in the code for optimization (e.g., loops, code sections), filter them based on their attributes (e.g., the loop variable) and then apply the transformation (e.g., loop tiling). It can also be used for programming autoparallelization strategies as presented in Arabnejad et al. [35], optimization-space exploration, and integration with third-party tools.

There are more general approaches for code analysis and transformation, such as term rewriting. Stratego/XT [59] and Rascal [60] describe transformations based on pattern-matching and rewrite rules that are applied over an abstract representation obtained from the grammar of the target language. The Clava compiler, on the other hand, uses Clang for parsing and analysis, and can be developed incrementally, adding code points, attributes, and actions as needed.

Several optimizing compilers also support transformations, such as auto-parallelization. For instance, Par4All [61] is an automatic parallelizing and optimizing compiler for C and Fortran, with backends for OpenMP, OpenCL, and CUDA. The auto-parallelization feature of the Intel Compiler[8] automatically detects loops that can be safely and efficiently executed in parallel and generates multi-threaded code of the input program. However, these approaches offer minimal control over the transformations they support and do not allow the kind of customization that Clava provides.

Overall, the use of the tools and approaches presented above can contribute to the performance engineering tasks and can also be seen as complementary options to support users in the complex stages of performance engineering. However, we believe that the Pegasus approach is unique and holistically unifies performance engineering tasks, genuinely contributing to productivity gains.

8 Conclusion

This paper presented Pegasus, an approach for the semi-automation of the tasks in a typical performance engineering methodology for software applications, and targeting high-performance computing (HPC) systems. The Pegasus approach is supported by a framework consisting of a source-to-source compiler (Clava), a domain-specific language (LARA), which allows developers and performance engineers to program strategies at different levels of the methodology, by a runtime Autotuner (mARGOt), and by libraries that contribute to the effectiveness of the approach.

We evaluated the Pegasus approach with core components of a futuristic navigation system case study targeting smart cities and requiring the use of an HPC platform. The experimental results strongly show the importance of the approach in different stages of the software development process. Moreover, the results show that Pegasus contributes to more efficient implementations, otherwise requiring manual efforts. Specifically, the addressed components of the navigation system were improved in terms of execution time reductions and energy consumption savings.

The software metrics collected indicate that our approach may significantly save programming and performance tuning time and contribute to time-to-solution reductions. The approach is useful for application analysis, use and analysis of the impact of compiler optimizations, identification of possible operating points and knobs, and synthesis and integration of a runtime autotuner. LARA strategies support all these steps, some of them with high levels of reusability. These strategies are automatically applied to the application source code, and thus have high potential to reduce the efforts of developers and performance engineers.

The planned future work includes further automation of several tasks of the methodology, especially the ones regarding the interface to other third-party tools and the selection of the strategies according to the application analysis and the performance and energy consumption requirements. Furthermore, targeting distributed and heterogeneous systems is on our plans. We already have some Clava libraries that help with the generation of code for computation offloading and other Clava libraries that already perform directive-based parallelization, and that could be adapted for heterogeneous architectures.

Acknowledgments

This work was partially funded by the ANTAREX project through the EU H2020 FET-HPC program under grant no. 671623. Pedro Pinto and João Bispo acknowledge the support provided by Fundação para a Ciência e a Tecnologia, Portugal under Ph.D. grant SFRH/BD/141783/2018 and Post-Doc grant SFRH/BPD/118211/2016, respectively.

References

[1] J. M. P. Cardoso, J. G. F. C. Coutinho, and P. C. Diniz, *Embedded Computing for High Performance: Efficient Mapping of Computations Using Customization, Code Transformations and Compilation*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2017.

[2] A. Dubey, S. R. Brandt, R. C. Brower, M. Giles, P. D. Hovland, D. Q. Lamb, F. Löffler, B. Norris, B. O'Shea, C. Rebbi, M. Snir, and R. Thakur, "Software abstractions and methodologies for hpc simulation codes on future architectures," *arXiv preprint arXiv:1309.1780*, 2013.

[3] P. Balaprakash, A. Tiwari, and S. M. Wild, "Multi objective optimization of hpc kernels for performance, power, and energy," in *International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*. Springer, 2013, pp. 239–260.

[4] R. Rabenseifner, "Hybrid parallel programming on hpc platforms," in *proceedings of the Fifth European Workshop on OpenMP, EWOMP*, vol. 3, 2003, pp. 185–194.

[8] Intel C++ Compiler. For more information, please visit https://software.intel.com/en-us/c-compilers/

[5] G. Oger, D. Le Touzé, D. Guibert, M. De Leffe, J. Biddiscombe, J. Soumagne, and J.-G. Piccinali, "On distributed memory mpi-based parallelization of sph codes in massive hpc context," *Computer Physics Communications*, vol. 200, pp. 1–14, 2016.

[6] M. Bauer, H. Cook, and B. Khailany, "Cudadma: optimizing gpu memory bandwidth via warp specialization," in *Proceedings of 2011 international conference for high performance computing, networking, storage and analysis*. ACM, 2011, p. 12.

[7] P. Balaprakash, J. J. Dongarra, T. Gamblin, M. Hall, J. K. Hollingsworth, B. Norris, and R. W. Vuduc, "Autotuning in high-performance computing applications," *Proceedings of the IEEE*, vol. 106, pp. 2068–2083, 2018.

[8] J. M. Cardoso, T. Carvalho, J. G. Coutinho, W. Luk, R. Nobre, P. Diniz, and Z. Petrov, "Lara: An aspect-oriented programming language for embedded systems," in *Proceedings of the 11th Annual International Conference on Aspect-oriented Software Development*, ser. AOSD '12. New York, NY, USA: ACM, 2012, pp. 179–190.

[9] J. M. P. Cardoso, J. G. F. Coutinho, T. Carvalho, P. C. Diniz, Z. Petrov, W. Luk, and F. Gonçalves, "Performance-driven instrumentation and mapping strategies using the lara aspect-oriented programming approach," *Software: Practice and Experience*, vol. 46, no. 2, pp. 251–287, 2016.

[10] P. Pinto, T. Carvalho, J. Bispo, M. A. Ramalho, and J. M. Cardoso, "Aspect composition for multiple target languages using lara," *Computer Languages, Systems & Structures*, vol. 53, pp. 1–26, 2018.

[11] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," in *ECOOP'97 – Object-Oriented Programming*, M. Akşit and S. Matsuoka, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 220–242.

[12] J. M. Cardoso, T. Carvalho, J. G. Coutinho, R. Nobre, R. Nane, P. C. Diniz, Z. Petrov, W. Luk, and K. Bertels, "Controlling a complete hardware synthesis toolchain with lara aspects," *Microprocessors and Microsystems*, vol. 37, no. 8, Part C, pp. 1073 – 1089, 2013.

[13] C. Silvano, G. Agosta, S. Cherubin, D. Gadioli, G. Palermo, A. Bartolini, L. Benini, J. Martinovič, M. Palkovič, K. Slaninová, J. Bispo, J. M. P. Cardoso, R. Abreu, P. Pinto, C. Cavazzoni, N. Sanna, A. R. Beccari, R. Cmar, and E. Rohou, "The antarex approach to autotuning and adaptivity for energy efficient hpc systems," in *Proceedings of the ACM International Conference on Computing Frontiers*, ser. CF '16. New York, NY, USA: ACM, 2016, pp. 288–293.

[14] C. Silvano, G. Agosta, A. Bartolini, A. Beccari, L. Benini, L. Besnard, J. Bispo, R. Cmar, J. Cardoso, C. Cavazzoni, D. Cesarini, S. Cherubin, F. Ficarelli, D. Gadioli, M. Golasowski, I. Lasri, A. Libri, C. Manelfi, J. Martinovic, and E. Vitali, "Supporting the scale-up of high performance application to pre-exascale systems: The antarex approach," in *27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, 02 2019, pp. 116–123.

[15] C. Silvano, G. Agosta, A. Bartolini, A. R. Beccari, L. Benini, L. Besnard, J. Bispo, R. Cmar, J. M. Cardoso, C. Cavazzoni, D. Cesarini, S. Cherubin, F. Ficarelli, D. Gadioli, M. Golasowski, A. Libri, J. Martinovič, G. Palermo, P. Pinto, E. Rohou, K. Slaninová, and E. Vitali, "The antarex domain specific language for high performance computing," *Microprocessors and Microsystems*, vol. 68, pp. 58 – 73, 2019.

[16] D. Gadioli, E. Vitali, G. Palermo, and C. Silvano, "mARGOt: a Dynamic Autotuning Framework for Self-aware Approximate Computing," *IEEE Transactions on Computers*, 2018.

[17] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage, "Improving software diagnosability via log enhancement," *SIGARCH Comput. Archit. News*, vol. 39, no. 1, pp. 3–14, Mar. 2011.

[18] R. S. Arnold, "Software restructuring," *Proceedings of the IEEE*, vol. 77, no. 4, pp. 607–617, April 1989.

[19] W. G. Griswold and D. Notkin, "Automated assistance for program restructuring," *ACM Trans. Softw. Eng. Methodol.*, vol. 2, no. 3, pp. 228–269, Jul. 1993.

[20] T. Mens and T. Tourwe, "A survey of software refactoring," *IEEE Transactions on Software Engineering*, vol. 30, no. 2, pp. 126–139, Feb 2004.

[21] D. Dig, "A refactoring approach to parallelism," *IEEE Software*, vol. 28, no. 1, pp. 17–22, Jan 2011.

[22] E. Murphy-Hill, C. Parnin, and A. P. Black, "How we refactor, and how we know it," *IEEE Trans. Softw. Eng.*, vol. 38, no. 1, pp. 5–18, Jan. 2012.

[23] E. Tempero, T. Gorschek, and L. Angelis, "Barriers to refactoring," *Commun. ACM*, vol. 60, no. 10, pp. 54–61, Sep. 2017.

[24] G. H. Golub and C. F. Van Loan, *Matrix computations*. JHU press, 2012, vol. 3.

[25] P. Bürgisser, M. Clausen, and M. A. Shokrollahi, *Algebraic complexity theory*. Springer Science & Business Media, 2013, vol. 315.

[26] S. L. Graham, P. B. Kessler, and M. K. Mckusick, "Gprof: A call graph execution profiler," in *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, ser. SIGPLAN '82. New York, NY, USA: ACM, 1982, pp. 120–126.

[27] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '07. New York, NY, USA: ACM, 2007, pp. 89–100.

[28] A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. S. Müller, and W. E. Nagel, "The vampir performance analysis tool-set," in *Tools for High Performance Computing*, M. Resch, R. Keller, V. Himmler, B. Krammer, and A. Schulz, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 139–155.

[29] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanna, and C. Le, "RAPL: Memory Power Estimation and Capping," in *Proceedings of the 16th ACM/IEEE International Symposium on Low Power Electronics and Design*, ser. ISLPED 2010. New York, NY, USA: ACM, 2010, pp. 189–194.

[30] M. Wolfe, *High Performance Compilers for Parallel Com-*

puting, C. Shanklin and L. Ortega, Eds. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[31] A. Hartono, B. Norris, and P. Sadayappan, "Annotation-based empirical performance tuning using orio," in *2009 IEEE International Symposium on Parallel Distributed Processing*, May 2009, pp. 1–11.

[32] R. Chandra, *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[33] M. Li and X. Yao, "Quality evaluation of solution sets in multiobjective optimisation: A survey," *ACM Comput. Surv.*, vol. 52, no. 2, pp. 26:1–26:38, Mar. 2019.

[34] D. Michie, ""Memo" functions and machine learning," *Nature*, vol. 218, no. 5136, 1968.

[35] H. Arabnejad, J. Bispo, J. G. Barbosa, and J. M. P. Cardoso, "An OpenMP based Parallelization Compiler for C Applications," in *16th IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA 2018)*, Dec. 2018.

[36] H. Arabnejad, J. Bispo, J. M. P. Cardoso, and J. G. Barbosa, "Source-to-source compilation targeting openmp-based automatic parallelization of c applications," *The Journal of Supercomputing*, Dec 2019. [Online]. Available: https://doi.org/10.1007/s11227-019-03109-9

[37] D. Gadioli, R. Nobre, P. Pinto, E. Vitali, A. H. Ashouri, G. Palermo, J. M. P. Cardoso, and C. Silvano, "SOCRATES — A seamless online compiler and system runtime autotuning framework for energy-aware applications," in *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2018, pp. 1143–1146.

[38] M. Wolfe, "More iteration space tiling," in *Supercomputing '89: Proceedings of the 1989 ACM/IEEE Conference on Supercomputing*, Nov 1989, pp. 655–664.

[39] E. Vitali, D. Gadioli, G. Palermo, M. Golasowski, J. Bispo, P. Pinto, J. Martinovič, K. Slaninová, J. Cardoso, and C. Silvano, "An Efficient Monte Carlo - based Probabilistic Time-Dependent Routing Calculation Targeting a Server-Side Car Navigation System," *IEEE Transactions on Emerging Topics in Computing*, 2019.

[40] M. Golasowski, J. Beránek, M. Šurkovský, L. Rapant, D. Szturcová, J. Martinovič, and K. Slaninová, "Alternative paths reordering using probabilistic time-dependent routing," in *Advances in Networked-based Information Systems*, L. Barolli, H. Nishino, T. Enokido, and M. Takizawa, Eds. Springer International Publishing, 2020, pp. 235–246.

[41] I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck, "Alternative routes in road networks," *Journal of Experimental Algorithmics*, vol. 18, Apr. 2013.

[42] T. Chondrogiannis, P. Bouros, J. Gamper, and U. Leser, "Alternative routing: K-shortest paths with limited overlap," in *Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems*, ser. SIGSPATIAL '15. New York, NY, USA: Association for Computing Machinery, 2015.

[43] J. Hanzelka, M. Běloch, J. Křenek, J. Martinovič, and K. Slaninová, "Betweenness propagation," in *Computer Information Systems and Industrial Management*, K. Saeed and W. Homenda, Eds. Springer International Publishing, 2018, pp. 279–287.

[44] J. Hanzelka, M. Běloch, J. Martinovič, and K. Slaninová, "Vertex importance extension of betweenness centrality algorithm," in *Data Management, Analytics and Innovation*, V. E. Balas, N. Sharma, and A. Chakrabarti, Eds. Singapore: Springer Singapore, 2019, pp. 61–72.

[45] A. McLaughlin and D. A. Bader, "Accelerating gpu betweenness centrality," *Commun. ACM*, vol. 61, no. 8, pp. 85–92, Jul. 2018.

[46] X. Tian and K. Benkrid, "High-performance quasi-monte carlo financial simulation: Fpga vs. gpp vs. gpu," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 3, no. 4, Nov. 2010.

[47] M. Golasowski, R. Tomis, J. Martinovič, K. Slaninová, and L. Rapant, "Performance Evaluation of Probabilistic Time-Dependent Travel Time Computation," in *Computer Information Systems and Industrial Management*, K. Saeed and W. Homenda, Eds. Springer International Publishing, 2016, pp. 377–388.

[48] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An overview of AspectJ," in *European Conference on Object-Oriented Programming*. Springer, 2001, pp. 327–354.

[49] O. Spinczyk, A. Gal, and W. Schröder-Preikschat, "AspectC++: An Aspect-oriented Extension to the C++ Programming Language," in *Proceedings of the Fortieth International Conference on Tools Pacific: Objects for Internet, Mobile and Embedded Applications*, ser. CRPIT '02. Darlinghurst, Australia, Australia: Australian Computer Society, Inc., 2002, pp. 53–60.

[50] H. Liu, X. Guo, and W. Shao, "Monitor-based instant software refactoring," *IEEE Trans. Softw. Eng.*, vol. 39, no. 8, pp. 1112–1126, Aug. 2013.

[51] R. Morales, R. Saborido, F. Khomh, F. Chicano, and G. Antoniol, "Earmo: An energy-aware refactoring approach for mobile apps," in *Proceedings of the 40th International Conference on Software Engineering*. New York, NY, USA: ACM, 2018, pp. 59–59.

[52] C. Dave, H. Bae, S. Min, S. Lee, R. Eigenmann, and S. Midkiff, "Cetus: A Source-to-Source Compiler Infrastructure for Multicores," *Computer*, vol. 42, no. 12, pp. 36–42, Dec 2009.

[53] D. Quinlan, "Rose: Compiler support for object-oriented frameworks," *Parallel Processing Letters*, vol. 10, no. 02n03, pp. 215–226, 2000.

[54] G. Rudy, M. M. Khan, M. Hall, C. Chen, and J. Chame, "A programming language interface to describe transformations and code generation," in *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 2010, pp. 136–150.

[55] M. Christen, O. Schenk, and H. Burkhart, "PATUS: A Code Generation and Autotuning Framework for Parallel Iterative Stencil Computations on Modern Microarchitectures," in *IEEE International Parallel & Distributed Processing Symposium (IPDPS)*. IEEE, May 2011, pp. 676–687.

[56] T. S. F. X. Teixeira, C. Ancourt, D. Padua, and W. Gropp, "Locus: A system and a language for program optimization," in *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO 2019. Piscataway, NJ, USA: IEEE Press, 2019, pp. 217–228.

[57] R. Keryell, C. Ancourt, F. Coelho, B. Eatrice, C. Frann, F. Irigoin, and P. Jouvelot, "Pips: a workbench for building interprocedural parallelizers, compilers and optimizers," École Nationale Supérieure des Mines de Paris, France., Tech. Rep., 04 1996.

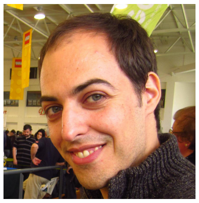
[58] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe, "Opentuner: An extensible framework for program autotuning," in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, ser. PACT '14. New York, NY, USA: ACM, 2014, pp. 303–316.

[59] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser, "Stratego/xt 0.17. a language and toolset for program transformation," *Science of Computer Programming*, vol. 72, no. 1-2, pp. 52 – 70, 2008.

[60] P. Klint, T. Van Der Storm, and J. Vinju, "Rascal: A domain specific language for source code analysis and manipulation," in *Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*. IEEE, 2009, pp. 168–177.

[61] N. Ventroux, T. Sassolas, A. Guerre, B. Creusillet, and R. Keryell, "Sesam/par4all: a tool for joint exploration of mpsoc architectures and dynamic dataflow code generation," in *Proceedings of the 2012 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*. ACM, 2012, pp. 9–16.

Pedro Pinto is a PhD student at the Faculty of Engineering of the University of Porto. Pedro Pinto obtained his MSc from the same institution in 2012. Since graduating, he has been involved in several research projects in the area of compilers. His main research interests include source-to-source compilation, application analysis and optimization, and code transformations, as well as broader topics such as programming languages, high-performance computing and machine learning.



João Bispo is a post-doctoral researcher at the SPeCS lab in the Faculty of Engineering, University of Porto (FEUP). He is doing research since the end of his bachelor's (2006), and in 2012 received the Ph.D. degree from Instituto Superior Técnico (IST), Lisbon, with a thesis about automatic runtime migration of binary code to hardware. His research interests an on hardware synthesis from high-level descriptions and source-to-source compilation.

João M. P. Cardoso got his Ph.D. degree in Electrical and Computer Engineering from the IST/UTL (Technical University of Lisbon), Lisbon, Portugal, in 2001. He is Full Professor at the Dep. of Informatics Eng., Faculty of Eng. of the University of Porto, and a senior researcher at INESC TEC. Before, he was with the IST/UTL (2006-2008), a senior researcher at INESC-ID (2001-2009), and with the University of Algarve (1993-2006). In 2001/2002, he worked for PACT XPP Technologies, Inc., Munich, Germany. He has been involved in the organization and served as a Program Committee member for many Int'l Conferences. He was co-scientific coordinator of the FP7-EU project REFLECT and technical manager of the H2020-EU project ANTAREX, and coordinator of various national funded projects. He has (co-)authored over 200 scientific publications. His research interests include compilation techniques, domain-specific languages, reconfigurable computing, high-level synthesis and application-specific architectures, and high-performance computing with an emphasis in embedded computing. He is a senior member of IEEE and ACM.

Jorge G. Barbosa received the BSc degree in Electrical and Computer Engineering from Faculty of Engineering of the University of Porto (FEUP), Portugal, the MSc in Digital Systems from University of Manchester Institute of Science and Technology, England, in 1993, and the PhD in Electrical and Computer Engineering from FEUP, Portugal, in 2001. Since 2001 he is an Assistant Professor at FEUP. His research interests are related to parallel and distributed computing, heterogeneous computing, scheduling in heterogeneous environments and cloud computing.

Davide Gadioli received his his Master of Science degree in Computer Engineering in 2013, while in 2019 he received the Ph.D degree in Computer Engineering, from Politecnico di Milano (Italy). Currently, he is a postdoc at Dipartimento dei Elettronica, Informazione e Bioingegneria (DEIB) of Politecnico di Milano. In 2015, he was a Visiting Student at IBM Research (The Netherlands). His main research interests are in application autotuning, autonomic computing and approximate computing.

Gianluca Palermo received his Master of Science degree in Electronic Engineering, in 2002, and the PhD degree in Computer Engineering, in 2006, from Politecnico di Milano (Italy). He is currently an Associate Professor at the Department of Electronics, Information and Bioengineering (DEIB) at the same University. Previously, he was Consultant Engineer at the Low-Power Design Group of AST - STMicroelectronics working on Network-on-Chip, and Research Assistant at the Advanced Learning and Research Institute (ALaRI) of the Universita' della Svizzera Italiana (Switzerland). His research interests include design methodologies and architectures for embedded and HPC systems focusing on autotuning aspects. He is an active member of the scientific community serving in organizing and program committees of several conferences in his research areas. Since 2003, he published more than 100 scientific papers in peer-reviewed conferences and journals. He is member of IEEE, ACM and HiPEAC.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2020.3001257, IEEE Transactions on Software Engineering

Jan Martinovič is currently Head of Advanced Data Analysis and Simulation Lab at IT4Innovations National Supercomputing Center, VSB – Technical University of Ostrava, Czech Republic. His research activities are focused on information retrieval, data processing, design and development of information systems and disaster management. His activities also cover a development HPC as a Service Middleware which allows to use HPC infrastructure remotely by specific API. Jan is coordinator of the H2020 ICT project LEXIS (Large-scale Execution for Industry & Society). He had previous experience with coordination of the different contracted research activities and had responsibility for the technical coordination of the several national projects. He was the Leader of IT4I as a partner of the two H2020-FETHPC-2014 projects ANTAREX and ExCAPE. He is also responsible for the research and development team of FLOREON+ system for disaster management support. He has published more than 100 papers in international journals and conferences.

Martin Golasowski is a researcher and a Ph.D. student in the Advanced Data Analysis and Simulation Laboratory of the IT4Innovations National Supercomputing Center of the Czech Republic. Topic of his research are high performance programming models for Monte Carlo methods and emerging heterogenous architectures. He participated in the H2020 FET project ANTAREX, H2020 ICT project LEXIS and in the research activities and development of FLOREON+ system for disaster management support. His other interests include parallel computing architectures, data processing and visualisation. He has published more than 30 conference papers and several journal articles.

Kateřina Slaninová is Deputy head of Advanced Data Analysis and Simulations Lab at IT4Innovations National Supercomputing Center, VSB – Technical University of Ostrava, Czech Republic. She has got a doctoral degree in Informatics from VSB – Technical University of Ostrava, Czech Republic. Kateřina research interests include information retrieval, traffic analysis, vehicle routing problem, hyperparameter search, data mining, process mining, and complex networks. Her recent activities also cover cooperation with SMEs in areas such as traffic management, artificial intelligence, time series analysis, etc. She participated in H2020 ICT project LEXIS and H2020 FETHPC project ANTAREX. She worked within the team of the Center for the Development of Transportation Systems RODOS. She has published more than 70 papers in international journals and conferences.

Radim Cmar is currently the solution architect at Sygic, Slovakia. His main expertise is in system modelling and architecture design for complex ITC systems such as mobile applications, server computing systems, and client to server communication systems. He graduated at Slovak Technical University of Bratislava in 1993. From 1997 to 2001 he was the research engineer at IMEC, Leuven, Belgium with the focus on HW/SW codesign methodologies for ASIC design. From 2001 to 2005 he was the system engineer at RFMD, San Jose, U.S. responsible for the design of the system-on-chip solutions for wireless communication. In 2007 he joined Sygic and since then he has been responsible for product definitions of various navigation components, and project lead for developing business solutions with many business partners. He participated in the H2020 FET project ANTAREX representing the industrial partner in the HPC solution for large scale navigation problems.

Cristina Silvano is a Full Professor of Computer Architectures at at the Department of Electronics, Information and Bioengineering (DEIB) of the Politecnico di Milano, Italy. Her main research interests are in energy-efficient embedded systems, design space exploration of manycore architectures and application autotuning for HPC. She has published more that 160 scientific papers in peer-reviewed journals and conferences, five books and she holds several patents in collaboration with Group Bull and STMicroelectronics. She was Project Coordinator of three European projects: H2020-ANTAREX, FP7-2PARMA and FP7-MULTICUBE. She has served in the organizing and program committees of several major conferences in computer architectures, embedded systems and electronic design automation. She is Associate Editor of ACM TACO and IEEE TC. She served as Independent Expert Reviewer for the European Commission and for several science foundations. In 2017, she has been elevated to the grade of IEEE Fellow.