International Journal of
HIGH PERFORMANCE
COMPUTING APPLICATIONS

Special Issue Paper

The International Journal of High Performance Computing Applications
1–17
© The Author(s) 2019
Article reuse guidelines:
sagepub.com/journals-permissions
DOI: 10.1177/1094342019865606
journals.sagepub.com/home/hpc
SAGE

Managing code transformations for better performance portability

Thiago SFX Teixeira, William Gropp and David Padua

Abstract
Code optimization is an intricate task that is getting more complex as computing systems evolve. Managing the program optimization process, including the implementation and evaluation of code variants, is tedious, inefficient, and errors are likely to be introduced in the process. Moreover, because each platform typically requires a different sequence of transformations to fully harness its computing power, the optimization process complexity grows as new platforms are adopted. To address these issues, systems and frameworks have been proposed to automate the code optimization process. They, however, have not been widely adopted and are primarily used by experts with deep knowledge about underlying architecture and compiler intricacies. This article describes the requirements that we believe necessary for making automatic performance tuning more broadly used, especially in complex, long-lived high-performance computing applications. Besides discussing limitations of current systems and strategies to overcome these, we describe the design of a system that is able to semi-automatically generate efficient platform-specific code. In the proposed system, the code optimization is programmer-guided, separately from application code, on an external file in what we call *optimization programming*. The language to program the optimization process is able to represent complex collections of transformations and, as a result, generate efficient platform-specific code. A database manages different optimized versions of code regions, providing a pragmatic approach to performance portability, and the framework itself has separate components, allowing the optimized code to be used on systems without installing all of the modules required for the code generation. We present experiments on two different platforms to illustrate the generation of efficient platform-specific code that performs comparable to hand-optimized, vendor-provided code.

Keywords
Code generation, optimization, compilers, domain-specific language, high-performance computing

1. Introduction

The push for more performance has increased the complexity of hardware platforms over time. It is the reason behind the addition of new features and the development of accelerators. New memory technologies, deep cache hierarchies, branch prediction, out-of-order execution, and speculative execution complicate the use and performance modeling of these highly complex platforms. They make programming harder and performance less predictable. Moreover, each hardware platform commonly requires a different sequence of optimizations to attain a high fraction of its nominal peak speed. Software developers must devote significant time to benefit from the computing power of modern CPUs and accelerators as the gap between the performance of hand-tuned and compiler-generated code has grown substantially.

As platforms evolve and new ones are adopted, programs must often be altered by the numerous optimizations needed for each target environment (hardware and software stack) to approximate maximum computing power. As a result, the code becomes unrecognizable over time, hard to maintain, and challenging to modify. Furthermore, as the code evolves, it is hard to keep the optimizations up to date. The need to develop and maintain separate versions of the application for each of the target platforms is an immense undertaking, especially for the large and long-lived applications commonly found in the high-performance computing (HPC) community.

An application code is portable if it runs on a diverse set of platforms without needing significant modifications and

Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois, USA

Corresponding author:
Thiago SFX Teixeira, Department of Computer Science, University of Illinois at Urbana-Champaign, Teixeira, 201 North Goodwin Avenue, Urbana, IL, USA.
Email: tteixei2@illinois.edu

produces a similar output. Ideally, the application code would also be *performance portable* and achieve high performance across a variety of platforms (Pennycook et al., 2016; Wolfe, 2016).

However, creating performance portable code is difficult as the optimization space is very large, and there are many important decisions that affect application efficiency on the target machines. These include not only the order of operations within large loops but also the choice and layout of data structure. The optimal or near-optimal choices often differ depending on the target platform.

The selection of architectures and algorithms and their correspondence is crucial to attaining high performance. This selection is not straightforward; it can span from using specific algorithms selected for one particular architecture to selecting very general algorithms developed without any architecture insight. In the former case, high performance is achieved at the expense of portability given that it is so specific that it cannot execute on different architectures. In the latter case, the portability comes at the expense of performance, because without any insight on the architecture, it is very difficult to efficiently exploit the resources available.

Selecting the algorithm among many to solve a specific problem is the first and foremost decision. It also highly depends on the target machine and often on the problem size and other characteristics of the input. In some cases, multiple fine-grained algorithmic selection can be very advantageous as shown by Ansel et al. (2009). This composition complicates even more the algorithm selection task given the combinatorial explosion of possibilities. For instance, the optimal sorting method would use different algorithms based on the number of elements to be sorted. For very few elements, the insertion sort is faster, whereas for medium number of elements, the quicksort is faster; for very large inputs, the radix sort results in better performance. It is also possible to have a composition among the three different algorithms, once quick sort and radix sort recursively decomposes the problem in subproblems until it is small enough to apply the insertion sort (Li et al., 2004).

Choosing the data structure appropriately may increase data locality and reduce cache misses. Data placement in a multidimensional array can affect performance depending how the array is linearized in memory. For some applications, converting the data between different structures on the fly may improve performance, despite the cost of the conversion.

The use of appropriate compiler flags typically leads to a speedup without a significant increase in compilation time. The flags are applicable to the whole code and the heuristics used by compilers to apply transformations do not guarantee improvement in all cases. Notably, architecture-specific compiler flags have a higher chance of better results, because the heuristics can assume a narrower scenario. The use of compiler directives or pragmas is also an interesting approach for isolating the optimizations to specific code regions where performance is crucial. Nonetheless, these directives are not standardized across compilers, leaving the code less portable. The results achieved by selecting compiler flags and directives rarely remain the same for new processors, requiring the experiments to be performed all over every time a new one is released.

We rely on an approach that decouples the performance expert role from the application expert role (separation of concerns). The *baseline version* defined by the developer should be as readable as possible and avoid any platform- or compiler-specific optimizations. The application of the transformations to the baseline is controlled from an external file in what we refer as *optimization programming*. This approach allows the use of architecture-specific optimizations while keeping the code maintainable in the long term. A database of different variants for different architectures (and possibly different problem sizes and other input parameters) is also maintained.

There have been a number of tools to ease the burden of the optimization process on the programmer. They commonly require manual refactoring (Ansel et al., 2009; Fatahalian et al., 2006; Hartono et al., 2009) and provide little control over the steps being carried out. They also are not prepared to coexist with other tools and cannot be incrementally adopted (Basili et al., 2008).

In the following list, we describe what we believe are the requirements for making autotuning more accepted in applications. These requirements are based on our experiences with several applications, and in particular with a large, complex multi-physics application, being developed as part of the Center for the Exascale Simulation of Plasma-Coupled Combustion (XPACC, 2018):

1. A straightforward, clean version of the code that is understandable by the computational scientist and program developer. This is the *baseline* code and is the one that may be modified by the code developers.
2. The code should run in the absence of any tool, so that the developers are comfortable that their code will run even if the system fails for some reason.
3. A clean way to provide extra semantic information is needed; for example, this might be to indicate that a loop is short or long or that a set of functions is always called in the same order.
4. Code must run with good performance on multiple platforms and architectures, at least as long as the same algorithm is appropriate.
5. Because in practice code tuning and optimization is difficult to make fully automatic, there needs to be a way for a performance expert to provide additional, possibly target-specific, information about optimizations.
6. Because autotuning can be expensive (as many versions may need to be examined), the system must store the results of the autotuning step(s) and use previously found optimized code whenever possible.

7. Changes to the baseline code should ensure that "stale" optimized versions of the code are not used and preferably replaced by updated versions.
8. Hand-tuned optimizations should be allowed.
9. Using (as opposed to creating) the optimized code *must not* require installing the code generation and autotuning frameworks, as these are often difficult to install and use on many platforms.
10. The system should make it possible to gather performance data from a remote system, permitting the framework to run on a system on which the autotuning and code transformation tools can be installed.

From these requirements, we made the following design decisions:

- Use of annotated code, written in C, C++, or Fortran, with high-level information that marks regions of code for optimization (addresses 1 and 2).
- Use annotations that only cover high-level, platform-independent information (addresses 3).
- Maintain platform and tool-dependent information (e.g. loop-unroll depth) in a separate *optimization file* (addresses 5).
- Maintain a database of optimized code, organized by target platform and other parameters (addresses 4 and 6).
- Maintain, in the database, a hash of the relevant parts of the code for each transformed section, and this hash is confirmed before making use of a previously stored autotuned version (addresses 7).
- Allow the insertion of hand-tuned versions of the code into the database so that they can be used by the system in the same way that code generated by the autotuning step (addresses 5 and 8).
- Separate the steps of determining optimized code and populating the database from extracting code from the database to replace labeled code regions in the baseline version (addresses 9).
- Provide some support for running variants on a remote system, meaning that the full system need not be installed on the target system; this is especially important when the target is a supercomputer (addresses 9 and 10).
- Allow the inclusion and use of a preferred, custom version. For some applications and libraries, there is already a preferred, hand-optimized version (e.g. many of the kernel operations in portable, extensible toolkit for Scientific computation [PETSc] have manually unrolled loops; Balay et al., 2018a, 1997). A small change to our approach accommodates this by placing the *baseline* version into the database; this is the version used by the autotuning tools (addresses 2). We have not implemented this option, but it is an important part of the design and do plan to implement it in the future.

Not included above is the integrated support for debugging code; that is, methods that relate the autotuned code directly to the original baseline code in a way that can be presented by a debugger. While desirable and beneficial, we do not view this as essential, in part because at high levels of optimizations, compilers often perform complex code transformations that are not well reflected in the source code that a debugger may present. While this does complicate debugging, it is not a new problem. In fact, because our system stores both the baseline code and the transformed code (created through source-to-source transformations), it should be *easier* to debug code in this approach than relying on transformations handled entirely within a compiler.

1.1. Our approach

We implemented these design decisions in Locus (Teixeira et al., 2019). Locus is a semi-automatic approach to assist performance experts and code developers in the performance optimization process of programs developed in mainstream programming languages (C, C++, and Fortran). The system is nonprescriptive, which means that if none of the optimizations can be applied or improve performance, the baseline (original version) is used instead.

Regions of interest in the baseline version are marked and given an identifier. The optimization program uses this identifier to specify where to apply each transformation. Multiple regions with the same identifier will receive the same optimization steps (not necessarily the same resulting code).

Locus combines expert knowledge with empirical search, automates much of the optimization process, and gives total control to the developers. The system defines an interface to use external transformation and search modules. The idea is to have a collaborative environment where existing modules can be integrated in a single system. This enables the comparison among modules and the selection of the one that generates the most performant code.

Locus does not require the installation of any specific module. Only the ones used require installation. The full toolset integrated in Locus, however, requires a long and cumbersome installation process that is hard to replicate in all target machines. Therefore, the Locus system uses a database of platform-specific variants that separates the code generation from the uses of the generated optimized code. It also has support for a remote empirical search in which the driver that traverses the optimization space and the code generation are executed on a different machine than the target one where the variants are assessed.

The main contributions of this article are:

- a system able to generate, assess, and manage a database of platform-specific code variants for different code regions that separates code generation from their uses;

Table 1. Automatic program optimization approaches.

	Domain	Optimization domain	Optimization time	Search method
High-level abstractions				
SPIRAL	Signal processing	Rewriting rules	Offline	Dynamic programming, evolutionary (+others)
Lift	General purpose	Rewriting rules	Offline	Bandit
Halide	Image processing	Scheduling	Offline	Stochastic
Pochoir	Stencils	Cache-oblivious algorithm	Offline	—
Nonprogrammable				
FFTW	Fast Fourier transform	Combination of solvers for FFTs	Offline	Dynamic programming
Atlas	Dense linear algebra	Blocking, scheduling, and unrolling	Offline	Exhaustive
OSKI	Sparse linear algebra	Sparse solvers	Online	Heuristic
PHiPAC	Matrix multiplication	Adaptive library generator	Offline	Heuristic
Code transformations				
CHiLL	General purpose	Loop transformations	Offline	—
Pluto	General purpose	Loop Transformations	Offline	—
POET	General purpose	Parameterized code transformations	Offline	—
Orio	General purpose	Loop transformations	Offline	Nelder-Mead, simulated annealing
X Language	General purpose	Loop transformations	Offline	Exhaustive
Custom languages or languages extensions				
Sequoia	General purpose	Memory hierarchy aware language	Offline	—
Petabricks	General purpose	Algorithmic choices	Offline	Evolutionary
Kokkos	General purpose	Abstractions for parallel execution and data management	Offline	—
RAJA	General purpose	Abstractions for loops and data layouts	Offline	—
Alternative selection				
Nitro	General purpose	Variant selection	Offline, online	Classification (SVM)
Active Harmony	General purpose	Parametric traversal	Online	Nelder-Mead
OpenTuner	General purpose	Parametric traversal	Offline	AUC Bandit

- a distributed empirical search to populate the database that separates the variants code generation and their assessment;
- an evaluation of the approach in two different platforms: Intel x86 and IBM Power; and
- an evaluation of the empirical search using a fixed initial configuration instead of a random one (the default). The best variant found on Intel x86 was used as the initial configuration for the search on the IBM Power. Locus syntax greatly facilitates the definition of the initial configuration of the search.

The next section provides an overview of the related work. The design and implementation of our system for managing code transformations are presented next. After that we present experimental evaluation. We conclude and talk about future work in the last section.

2. Related work

Optimizing for performance is very dependent on the target platform as well as the problem domain, which makes the creation of one-solution-fits-all code extremely complicated.

There have been a number of projects to develop new programming models, languages, and tools aimed at providing programmers with productive means for achieving performance portability.

As shown in Table 1, we classify the approaches into High-level abstractions, Non-Programmable, Code Transformations, Custom Languages or Languages Extensions and Alternative Selection. The high-level programming abstractions tools provide a limited set of abstractions for specific domains to represent algorithms. From these representations, the tools are able to generate optimized platform-specific binaries. Steuwer et al. (2015) propose an approach in which the programmer writes a high-level expression composed of algorithmic primitives, and using rewriting rules, they map this high-level expression into a low-level expression in OpenCL. Halide (Ragan-Kelley et al., 2017) is a domain-specific language for complex image processing pipelines that is able to decouple algorithm representation from the schedule of the operations. The Pochoir stencil compiler (Tang et al., 2011) allows a programmer to write simple functional specifications for stencils that are translated into highly optimized implementation. SPIRAL (Püschel et al., 2005) includes a high-level mathematical framework that provides the link between the "high" mathematical level of transform algorithms and the "low" level of their code implementations. It features five search methods to select among the low-level options: Exhaustive, Random, Dynamic Programming, Evolutionary, and Hill Climbing.

The nonprogrammable approaches differ from the other approaches in that it does not start with user-written code, but instead the code is generated directly. These tools carry out all the tuning process without user intervention. It is completely automatic and typically uses heuristics to accelerate the search of the space of variants. These tools have a very specific domain, are self-contained, and are used as libraries by the applications.

For instance, FFTW (Frigo, 1999) is a comprehensive collection of fast C routines for computing the discrete Fourier transform (DFT). It does not implement a single DFT algorithm, but it is structured as a library of routines that can be composed in many ways, namely a plan. The plan dictates which routines should be executed and in what order taking into account the input size and type and which routines happens to be faster on the underlying hardware. ATLAS (Whaley and Dongarra, 1998) presents a methodology for the automatic generation of highly efficient basic linear algebra routines in different architectures. It isolates the machine-specific features of the operation to several routines, all of which deal with generating an optimized matrix multiplication that fit in the fastest level cache. This optimized routine is automatically created by the generator and uses timings to select the others parameters, such as block and loop unrolling factors.

The goal of PHiPAC (Bilmes et al., 1997) is to produce high-performance linear algebra libraries for a wide range of systems with minimum effort. The authors developed parameterized generators that produce code according to guidelines from a generic model of a set of C compilers and microprocessors. They also created scripts to automatically tune code for a particular system by varying generators' parameters.

OSKI (Vuduc et al., 2005) is a collection of low-level primitives that are integrated into automatically tuned computational kernels on sparse matrices. Differently from the previous approaches, where the tuning takes place offline, OSKI defers the tuning until the runtime to make decisions of the data structures and code transformations based on the input matrix and the underlying hardware.

The code transformation tools take as input an initial version of the code and the transformations to apply to it. This specification of transformations can take the form of annotations on the code (Donadio et al., 2006; Hartono et al., 2009), scripts with commands that represent each transformation (Chen et al., 2008), or command-line parameters. In some tools, the developer can implement their own program transformations (Yi et al., 2007).

CHiLL (Chen et al., 2008) contains loop transformation and code generation primitives. It takes as input the original code and a transformation script with bound parameters and generates a collection of code versions. POET (Yi et al., 2007) is an embedded scripting language for parameterizing complex code transformations so that they can be empirically tuned. The POET language is designed to decouple the empirical tuning aspect of performance optimization from the specifics of any library or compiler.

Another example in this class is Orio (Hartono et al., 2009), an annotation-based empirical performance-tuning system that takes annotated C source code as input, generates code variants of the annotated code, and empirically evaluates the performance of the generated codes, ultimately selecting the best-performing version to use for production runs. The X Language (Donadio et al., 2006) provides pragmas that can perform loop transformations and code transformations defined as pattern-replacement rules. Pluto (Bondhugula et al., 2008) automatically generates tiled code for efficient parallelism and locality through an affine transformation framework.

Custom languages and language extensions have been proposed with the goal of providing abstractions that insulate algorithmic choices, loop patterns, and data layout from the underlying platform. Sequoia's (Fatahalian et al., 2006) programming model assists the programmer in structuring bandwidth efficient parallel programs that remain easily portable to new machines. The abstraction of tasks are used as self-contained units of computation isolated in their own local address space, which helps express parallelism within a hierarchical organization.

Another programming language along with a compiler is PetaBricks (Ansel et al., 2009). The language incorporates fine-grained algorithmic choices in program optimization and allows the specification of different granularity and corner cases. The autotuner uses a choice dependency graph that contains the choices for computing each task and also encodes the implications of different choices on dependencies. It also contains a dynamic task scheduler and a runtime library to manage reading, writing, and managing, inputs, outputs, and configurations.

RAJA (Hornung and Keasler, 2014) provides portable abstractions for loops that include loop transformations, reductions, scans, atomic operations, data layouts, and views. Loop bodies and traversals are decoupled via lambda expressions (loop bodies) and templates (loop traversal methods) and support execution policies for different programming model back ends. Kokkos (Edwards et al., 2014) library unifies abstractions for both fine-grain data parallelism and memory access patterns for performance portability manycore architectures.

The alternative selection tools assume that either the user or some other tool provides as input a collection of variants. The main concern of the designer of tools in this class is the selection process. As the search space is very large, the use of efficient techniques on selecting the best combination of code variants or parameters can drastically reduce the search time. Some of these tools make the decision online, whereas others have it defined offline. The advantage of making the decision online is that the input features can be used in the decision process (Muralidharan et al., 2014). The design of languages, compilers, and runtime systems for the development and selection of algorithmic variants has shown to be a successful way of optimizing irregular iterative and recursive problems on parallel platforms (Ansel et al., 2009; Fatahalian et al., 2006).

The Nitro (Muralidharan et al., 2014) framework focuses on how code variants and meta-information for variant selection are expressed and uses a classification technique to select the most appropriate variant during application's runtime. The codes are represented through library calls rather than language extensions. Input features that significantly affect the variant selection can be calculated by providing functions to the system. A two-phase approach is used. First, the application is processed to generate a model over the code variants that are also provided by the user. Second, during execution, the production version uses the model generated in the first phase and adapts the execution based on input data.

Active Harmony (Chung and Hollingsworth, 2004) permits application programmers to express application-level parameters and automates the process of searching among a set of alternative implementations. It has been combined with CHiLL (Tiwari et al., 2011) to search for the best sequence of loop transformation variants of computationally intensive kernels. The OpenTuner (Ansel et al., 2014) project presents a new framework for building domain-specific program autotuners. It features an extensible configuration and technique representation able to support complex and user-defined data types and custom search heuristics.

Some systems attempt to separate the role of the performance tuning expert from the application domain. This separation can allow the programmer to focus on application issues and performance tuning at different points in time. Or, if they are not the same person, make these two tasks more independent and more productive, once each expert focus on their own domain. Besides, multiple tuning specifications can be generated and tested for different architectures, making the application performance portable.

Sequoia, for instance, maintains a strict separation between the algorithm implementation and the machine-specific optimizations. The autotuning system in Petabricks outputs an application configuration file containing the choices selected. This file can be either used to run the application, or it can be used by the compiler to build a binary with hard-coded choices. CHiLL also has an external file containing all the transformations to be applied to the code, which can be different depending on the machine specifics.

Another interesting feature of some of these systems is the automated validation of code variants. Petabricks has an automated consistency checking to make sure that the different algorithms solving the same problem produce consistent results. This helps the user to automatically detect bugs and increase confidence in correctness. Orio also supports an automated validation by comparing numerical results of the multiple transformed versions. This technique is not provably correct but provides a good testing coverage. It also compliments techniques that are provably correct, as these proofs do not extend to the implementation of the compilers, runtime systems, or hardware, and in real systems, users must confront faults and errors in these components as well.

3. Design and implementation

The program optimization process for multiple platforms is a complex undertaking. It requires a detailed understanding of the target platforms and the application being optimized. Our system facilitates for experts the application of optimizations and allows nonexperts to apply generic sequences of optmizations to find faster code. It also makes it easier for nonexperts to reuse results from others by sharing optimization sequences or the database of variants.

The requirements for such system are maintain a clean baseline version that executes on all target platforms; coexistence with other tools and frameworks; reuse modules already developed; abstract the use of transformation and search modules, so that it does not depend on any specific module; provide total control to the developer; and enable incremental use by providing localized optimization of the baseline.

The system fulfills these requirements by having a separate language for optimization programming; clearly identifying on the optimization program the steps to be carried out by the system; using source-to-source transformations to compose complex transformation sequences to harness the power of existing tools; having an interface able to integrate multiple transformation and search modules and abstract their use; and having a database of variants to reuse search results without installing the whole toolset required to generate those codes.

In the following sections, we describe our approach and the Locus implementation that separates the optimizations from the application code and creates a database of variants.

3.1. Locus system and language

Locus makes use of a powerful language for programming the optimizations and for representing complex optimization spaces. The software developer, through the Locus program, has complete control over the optimization sequences to be attempted to improve the performance.

The Locus language is dynamically typed and the system includes a translator and optimizer for it. At system's run-time, the resulting high-level representation of the optimization program is interpreted to generate variants of the baseline source code. An example of an optimization program is shown in Figure 1.

The programmer defines code regions in the source code using C pragmas or Fortran comments. There are two types of annotations: block and loop. The block annotation

```
dim=4096;
Search {
  buildcmd = "make␣clean␣all";
  runcmd = "./matmul";
}
CodeReg matmul {
  perm = permutation([0,1,2], init=[0,2,1]);
  RoseLocus.Interchange(order=perm);
  tileI = poweroftwo(2..dim, init=512);
  tileK = poweroftwo(2..dim, init=128);
  tileJ = poweroftwo(2..dim, init=2048);
  Pips.Tiling(loop="0", factor=[tileI, tileK, tileJ]);
  tileI_2 = poweroftwo(2..tileI, init=256);
  tileK_2 = poweroftwo(2..tileK, init=32);
  tileJ_2 = poweroftwo(2..tileJ, init=32);
  Pips.Tiling(loop="0.0.0.0",
              factor=[tileI_2, tileK_2, tileJ_2]);
  {
    None;
  } OR {
    tileI_3 = poweroftwo(2..tileI_2);
    tileK_3 = poweroftwo(2..tileK_2);
    tileJ_3 = poweroftwo(2..tileJ_2);
    Pips.Tiling(loop="0.0.0.0.0.0.0",
                factor=[tileI_3, tileK_3, tileJ_3]);
  }
}
```

Figure 1. Locus program for optimizing double-precision matrix–matrix multiplication (DGEMM). This program applies loop interchange, and a two- or three-level hierarchical tiling. It also provides the values to be used as the initial configuration for the search process.

indicates the begin and end of the code region. The loop annotation applies to loop nests. The loop annotations are for loop transformations, and block annotations are for alternative algorithm selection and optimizations comprising multiple code regions.

The annotations include a label that can be referenced in the optimization program. A *CodeReg NAME* statement in the optimization program precedes the set of statements that operate on the code regions in the source code labeled *NAME*. An *OptSeq NAME* precedes a set of statements that can be invoked within *CodeReg*s. An *OptSeq* can also be invoked by other *OptSeq*s. The operations that transform or get information from the source code can only be invoked from *CodeReg* and *OptSeq*, because only they are able to refer to the target code. The operations that extract information from the source code, namely *Query*, are used to decide which optimizatons to take next based on the current state of the code.

Operations in Locus are implemented in modules that are either external (e.g. RoseLocus) or intrinsic (e.g. Builtin). External modules are part of other systems and intrinsic modules are those developed specially for Locus.

The Locus language has special constructs that are useful to represent complex optimization spaces. The constructs are:

- OR blocks,
- OR statements,
- Optional statements, and
- datatypes for ranges of values (e.g. *enum*, *integer*, *float* and *poweroftwo*).

```
int main()
{
    int i, j, k;
    double t_start, t_end;
    init_array();
    t_start = rtclock();

#pragma @LOCUS loop=matmul
    for(i=0; i<M; i++)
      for(j=0; j<N; j++)
        for(k=0; k<K; k++)
    C[i][j] = beta*C[i][j] + alpha*A[i][k] * B[k][j];

    t_end = rtclock();
    print_array();
    printf("Time(ms)␣=␣%7.5lf\n", t_end-t_start);
    return 0;
}
```

Figure 2. On the left-hand side, the steps to add platform-specific optimized variants to the database after the search process. A direct Locus optimization program that loads the platform-specific variants is also automatically created for future use. During the deployment, shown on the right-hand side, the direct program accesses the database, retrieves the best variants for each code region, and replaces the ones in the baseline with the variants from the database.

The *OR* is used between sets of statements (blocks of code) to describe alternative transformations. Any statement with a preceding *is an optional statement that may or may not execute. The search module will decide during the search process which of the *OR* blocks and whether the optional statements will execute.

There are two kinds of optimization programs: *direct* and *search*. The *direct Locus program* contains a sequence of transformations with no optional statements, OR blocks, or ranges of values. The *search Locus program* contains search constructs to explore an optimization space. Many direct Locus programs can derive from a search Locus program. The Locus interpreter that carries out the transformation sequences only accepts as input direct Locus programs.

Figure 2 depicts the database construction process and its use. The database construction on the upper left-hand side of the figure represents the *search* workflow that starts with the conversion of the optimization space from the search Locus program to the notation accepted by the search module. After the conversion, the search process may suggest variants to be evaluated first or let the search module select these variants automatically. For each variant, a direct Locus program is created by replacing all of the search constructs with the specific values selected by the search module. Once the direct Locus program is created, the *direct* workflow (bottom-left horizontal transformation) uses it to generate a variant of the code (the *Optimized version* in the figure) and finally evaluate it according to some metric. The most common metric is the execution time, but the assessment can be customized to use any metric selected by the user (e.g. energy consumption).

At the end of this iterative process, the sequence of transformations used to generate the best variant is saved as a direct Locus program. The search final output can be shipped with the source code to be used in future deployments on the same platform. In this manner, the time spent on the search process, which can be very long depending on the size of the optimization space, is amortized among multiple users. The search process can limit the number of variants to be assessed or stop the search after a given elapsed time or when a metric goal is reached.

Along with the direct Locus program, the best variant code regions are saved in the database for later use. As depicted on the right-hand side of Figure 2, the search workflow final output (in direct Locus format) determines which code regions will be replaced and the shape of its replacement.

The variants assessed on the search workflow depend on the heuristics of the search module used. Our experience shows that the final result of the empirical process highly depends on the initial variant assessed, which is, if not provided, randomly selected. In other words, if the initial configuration performs badly, the search usually takes more time to get to a good solution.

The Locus syntax allows the specification of an initial value for each search construct. These values may be used as the initial configuration of the search process. The rules used to define the initial configuration are as follows:

- optional statements are assumed to be executed;
- ranges of values have an extra parameter, namely `init`), to provide the value;
- the first block of each OR block is used; and
- the first statement of each OR statement is used.

An example of annotated code is shown in Figure 3. The loop nest in the code with label *matmul* represents a matrix–matrix multiplication. This version is considered the baseline.

The optimization program in Figure 1 has a definition of a *CodeReg* with *NAME matmul*. This program first changes the loop order according to the permutation on the `perm` variable (e.g. from *ijk* to *ikj*) by calling *RoseLocus.Interchange*. Then, the loop nest is tiled twice by calling *Pips.Tiling* two times. In the end, there is an *OR* block that has the possibility of tiling again and generating a 3-level hierarchical tiling. Each tiling uses a range of values to cover different shapes, the best shape depends on the memory hierarchy, which is machine specific.

The code in Figure 1 also illustrates the representation of initial values that may be given to the search module to be assessed as the first variant. In this case, the initial values are given for the `perm` variable and for the variables `tileI`, `tileK`, `tileK`, `tileI_2`, `tileK_2`, and `tileJ_2`. The first variant will include the first block of the OR (only a `None`;) resulting in a two-level tiled variant.

In summary, Locus is invoked with the codes presented in Figures 1 and 3 and applies the optimizations defined in Figure 1 into the code of Figure 2. The result is the generation of multiples variants that are assessed in the target machine. One of these variants is shown in Figure 4. The variant code shown is an example of how complicated the variants generated can get. Locus is able to traverse the optimization space, generating and evaluating many variants automatically. Carrying out this process by hand is cumbersome, error-prone, and unproductive. Locus is valuable on the optimization process by allowing developers to concisely experiment with complex optimization spaces. Besides, different search techniques can be used with no changes to the optimization program.

```
int main()
{
    int i, j, k;
    double t_start, t_end;
    init_array();
    t_start = rtclock();

#pragma @LOCUS loop=matmul
    for(i_t = 0; i_t <= 7; i_t += 1)
    for(k_t = 0; k_t <= 3; k_t += 1)
    for(j_t = 0; j_t <= 1; j_t += 1)
    for(i_t_t = 8 * i_t;
        i_t_t <= ((8 * i_t) + 7); i_t_t += 1)
    for(k_t_t = 256 * k_t;
        k_t_t <= ((256 * k_t) + 255); k_t_t += 1)
    for(j_t_t = 32 * j_t;
        j_t_t <= ((32 * j_t) + 31); j_t_t += 1)
    for(i = 64 * i_t_t;
        i <= ((64 * i_t_t) + 63); i += 1)
    for(k = 4 * k_t_t;
        k <= ((4 * k_t_t) + 3); k += 1)
    for(j = 64 * j_t_t;
        j <= ((64 * j_t_t) + 63); j += 1)
    C[i][j] = beta*C[i][j] + alpha*A[i][k]*B[k][j];

    t_end = rtclock();
    print_array();
    printf("Time(ms)␣=␣%7.5lf\n", t_end-t_start);
    return 0;
}
```

Figure 3. Matrix–matrix multiplication (DGEMM) baseline version in C language.

```
dim=256;
Search {
  buildcmd = "make␣clean␣all";
  runcmd = "./heat3d";
}
CodeReg heat3d {
  tileJ = poweroftwo(2..dim);
  RoseUiuc.StripMine(loop=3, factor=tileJ);
  RoseUiuc.Interchange(order="0,2,1,3,4");
}
```

Figure 4. Code variant of the matrix–matrix multiplication generated by the optimization program in Figure 1. This code represents a two-level hierarchical tiling combined with loop reordering.

3.2. Operation and search modules

The system is able to integrate different operation and search modules. None of the modules, however, are required and they are assumed by Locus to be independent of each other. The availability of multiple and independent transformation modules provides a rich environment to transform applications. The abstraction of the optimization space provided by Locus allows the comparison of different search modules to best traverse the potentially large optimization space.

The operation modules include source-to-source transformations, queries to extract information from the code regions, methods to add pragmas (such as those used for OpenMP), and a method to replace the original code. External tools can be invoked by implementing the operation module interface. Operation modules can also be implemented internally by manipulating the Locus representation of the code. The collections of operation modules available are as follows:

- Pips: A source-to-source compilation framework for transforming C and Fortran 77 programs (Keryell et al., 1996). Locus has four loop transformations available from Pips (GenericTiling, fusion, unroll-and-jam, and unrolling).
- RoseLocus: An annotation-based source-to-source loop transformations developed by us using the Rose compiler infrastructure (Lidman et al., 2012).
- Pragmas: Using this module is possible to add pragmas, which can be compiler specific or for parallelization through OpenMP (Dagum and Menon, 1998). Examples of compiler-specific pragmas are `ivdep` and `vector always`, which can be used to enhance vectorization of the code generated by Intel ICC compiler.
- BuiltIn: includes queries to get information about loop nests, such as whether the nest is perfectly nested (*IsPerfectLoopNest*), and to get the nest depth (*LoopNestDepth*). It also includes a module to replace the original code with code snippets. The *Altdesc* is mostly used to incorporate hand-optimized kernels into an optimization sequence.

The search modules available are as folows:

- OpenTuner: A framework for building domain-specific program autotuners. It uses ensembles of search techniques that run at the same time, testing candidate configurations (Ansel et al., 2014). The search variables are represented in a flat optimization space. In this kind of representation, the search techniques are not aware of variables that influence the selection of other variables, which can worse the traversal results by taking into consideration points in the space that are not possible. Locus integration checks whether the selections suggested are valid before generating code and empirically evaluating the variant.
- HyperOpt: A framework for optimizing complex search spaces with real values, discrete, and conditional dimensions (Bergstra et al., 2013).

3.3. Database of variants

A database of platform-specific codes for different code regions that allows the use of pregenerated autotuned code is an important requirement for making autotuning more accessible. Despite the system's goal to facilitate the use of multiple transformation modules, in our experience (and especially the experience of our computational scientist users), the installation of the modules and their dependencies has shown to be complex and often not available for all systems.

As mentioned before, Locus allows the use of a direct optimization program to reuse results from previous empirical evaluations, which also makes the optimization results accessible to non-experts. The users of a direct optimization program, however, would still need to install all the modules invoked in the direct program. The database further facilitates the use of an direct optimization program, because the best variant found for each code region is saved along with a version of the direct Locus program resulted from the search process.

The database organizes information about the optimization process of an application. The process is commonly comprised of multiple independent empirical evaluations. The result of each evaluation is added to the database using a unique identifier. We assume that the evaluations on the same database are from the same baseline version and assessed on the same platform.

The best variant found for each code region is saved in the database. The code regions are indexed by the source code file that they belong, their Locus label, and an index value from 0 to the number of code regions in the source file. The index value is used to set apart code regions in case there are more than one in the same file with the same label. Identical code regions that have the same label will receive the same sequence of optimizations, but each one may have a different best variant saved on the database. Different codes can also have the same label and be applied the same sequence of optimizations; this is possible by using *Queries* on the optimization program that allows to customize the sequence of transformations according to the code region.

Each variant in the database has attached its attained metric (e.g. execution time). When a more recent empirical evaluation is conducted and finds a different variant that performs better, the current one in the database is replaced.

Figure 2 shows the database construction (left-hand side) and database use (right-hand side). The empirical evaluation results are added to the database after the search process. It also generates (or updates if there is one already) a direct Locus program that loads the variants from the database.

Changes in the baseline version may result the variants in the database incorrect. The database contains a hash of the baseline code region that each variant was generated from. The *BuiltIn.Altdesc* uses that hash to check whether the current code, which the variant is being incorporated

```
void heat3d(double A[2][N+2][N+2][N+2])
{
    int i, j, t, k;

#pragma @LOCUS loop=heat3d
    for(t = 0; t < T-1; t++) {
      for(i = 1; i < N+1; i++) {
        for(j = 1; j < N+1; j++) {
          for (k = 1; k < N+1; k++) {
A[(t+1)%2][i][j][k] = 0.125 * (A[t%2][i+1][j][k] -
  2.0 * A[t%2][i][j][k] + A[t%2][i-1][j][k])
  + 0.125 * (A[t%2][i][j+1][k] -
  2.0 * A[t%2][i][j][k] + A[t%2][i][j-1][k])
  + 0.125 * (A[t%2][i][j][k-1] -
  2.0 * A[t%2][i][j][k] + A[t%2][i][j][k+1])
  + A[t%2][i][j][k];
          }
        }
      }
    }
}
```

Figure 5. Locus program for optimizing the finite-difference solution to the 3-D heat equation. This program applies a tiling on the J loop.

into, is the same as the one the variant was generated from. The hash function may be language dependent; the current implementation removes all the whitespace characters before generating a SHA-1 key.

The empirical evaluation is often dependent on input parameters (e.g. problem size and matrices shape). The system allows the inclusion of tags that better represent the context in which the variants were generated. These tags can later be used to load the proper variants that will be deployed for similar contexts. The tags are provided as *key, value* pairs. For instance, the optimized code for matrix multiplication heavily depends on the shape of the matrices, which is one information that can be added as tags. During deployment, in case the database does not have an optimized code that matches exactly the *key, value* requested, a heuristic can be used to get the most appropriate from the available ones.

In our prototype implementation of the database, we use the file system to separate out the variants of different optimization processes. All the directories are automatically created and managed. As mentioned before, the variants are accessed from the database using a direct Locus program, which is also automatically updated after each empirical evaluation. The direct Locus program loads each code region in place using the *BuiltIn.Altdesc* module with the absolute path to where the variant is located as a parameter. Inside the database folder, there is one folder for each source file. Inside each source file's folder, there is a file for each code region containing the code representing the variant for it. The header of the file representing the variant contains the hash to check whether the variant is being applied into the same code region that it was generated from, and the metric attained by the variant. The name of the variant's file is the code region label, and its index followed by tags in alphabetical order. Figure 5 presents an example of the Locus database structure.

3.4. Distributed empirical search

In the same way that a database of pregenerated optimized code is important for making autotuning more accessible, the distributed empirical search allows the optimization of applications in platforms that do not have available all the code generation tools required. The transformation modules integrated into Locus are not available to all platforms. For instance, RoseLocus is based on Rose, and at the time of this publication, Rose was not available on IBM Power 9 Linux platforms.

One solution to assess platforms that lack the transformation modules required is to conduct the search process and code generation from a machine that has the necessary modules, move the generated code to the target platform, compile it there, and evaluate and return the metric so the search driver can decide which variant to evaluate next.

In search Locus programs, it is necessary to specify how to build, compile, and run the variants generated. We added to these commands the `scp` to copy the generated code, and `ssh` to invoke the compilation and running steps on the target platform. This strategy successfully allowed the evaluation of the IBM Power 9 Linux platform as presented in the next section.

4. Evaluation

We evaluated the performance of the code generated by Locus on two systems: IBM Power and Intel x86. The details of the systems are presented in Table 2. The IBM Power OS is a Linux Red Hat kernel version 4.14; the Intel x86 OS is a Linux Ubuntu kernel 4.4.0. We show optimization results on two benchmarks: a double-precision matrix–matrix multiplication as presented in Figure 3 using the optimization program shown in Figure 1 and a finite-difference solution to the 3-D heat equation presented in Figure 6 using the optimization program in Figure 7. The search process was conducted by OpenTuner and limited to the evaluation of 1000 variants or 5 h. The running time of variants evaluated was capped by the elapsed time of the best one found up to the moment the variant started its execution. This significantly reduced the total search time by limiting the execution of the bad variants that could last for hours.

The code generated on the IBM Power was compiled with XLC (version 16.1.1; flags `-O3` and `-qHot`) compiler and GNU GCC (version 8.2.0; flags `-O3`, `-mtune=native`, and `-ftree-vectorize`). For the Intel x86, the code was compiled with ICC (version 17.0.1; flags `-O3`, `-xHost`, `-ipo`, `-ansi-alias`, and `-fp-model precise`).

4.1. Matrix–matrix multiplication

The baseline code compiled with XLC performed significantly better than when compiled with GCC for the IBM Power platform. The XLC flag `-qhot` requests, according to IBM manual, high-order transformations such as loop

Table 2. Platforms used on the evaluation.

Platform	Processor	Clock	Cores	HT	L1	L2	L3	RAM
IBM Power	P9 8335-GTH	3.8 GHz	20	4	32 KB pr	512 KB sh	100 MB sh	570 GB
Intel x86	Xeon E5-2660 v3	2.60 GHz	10	2	32 KB pr	256 KB pr	25 MB sh	62 GB

Sh: shared; pr: private.

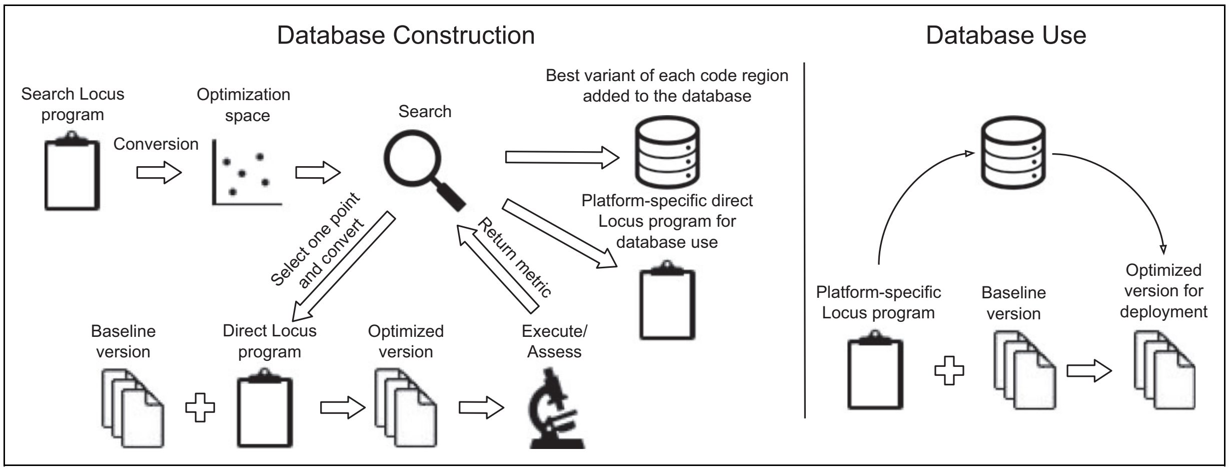
Database Construction
Database Use
Search Locus program
Conversion
Optimization space
Search
Best variant of each code region added to the database
Platform-specific direct Locus program for database use
Select one point and convert
Return metric
Baseline version
Direct Locus program
Optimized version
Execute/ Assess
Platform-specific Locus program
Baseline version
Optimized version for deployment

Figure 6. Baseline verion of the finite-difference solution to the 3-D heat equation in C language.

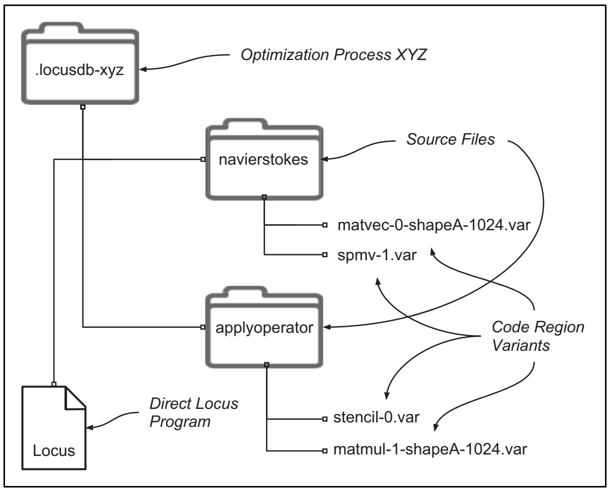

Figure 7. Locus database example. The top folder holds the variants for the optimization process named *XYZ*. It contains folders for the two source files that contain code regions that have been optimized—*navierstokes* and *applyoperator*. The optimized variants for these code regions are represented by the files with extension `.var`.

interchange, fusion, unrolling, and reduce the use of temporary arrays. Besides, by monitoring the compilation, we could see that XLC uses interprocedural analysis. As a consequence, the compilation time for XLC is at least three times longer than with GCC.

Moreover, the compiler[1] was able to detect that the baseline version was a matrix–matrix multiplication and replace it with a function call to a hand-optimized version. The version containing the invocation to the hand-optimized code is used as the reference on the performance evaluation. The compiler, however, was unable to detect matrix multiplication in the transformed code generated by Locus, which, in turn, was able to find variants that were faster than the XLC hand-optimized version.

Figure 8 shows results for IBM Power on the top and for Intel x86 on the bottom. On both platforms, a two- and three-level hierarchical tiling has been evaluated. The results include the best variant found by Locus and the baseline version using the two compilers. For the matrices of 2048 by 2048 shape, the best variant generated by Locus varied according to the compiler used. For XLC, the two-level tiling was the fastest, whereas with GCC, the three-level tiling attained the best performance. For matrices of 4096 by 4096 shape, both XLC and GCC found two-level tiling faster. For XLC, three-level tiling also attained performance close to that of the fastest two-level tiling version. Locus, however, could not find a three-level tiling variant, which when compiled with GCC was faster than the baseline.

For 8192 by 8192 matrices, due to the long time taken by most of the variants compiled with GCC, we only present results using the XLC compiler. In this case, the best performance is obtained with two-level tiling.

Search time is strictly dominated by the time to compile and run the code variants. Compilation is often fast, except for the case mentioned in which XLC runs high-order transformations and interprocedural analysis. The variant execution, however, depends on the matrix size and can take up to 30 min, since each variant was executed five times.

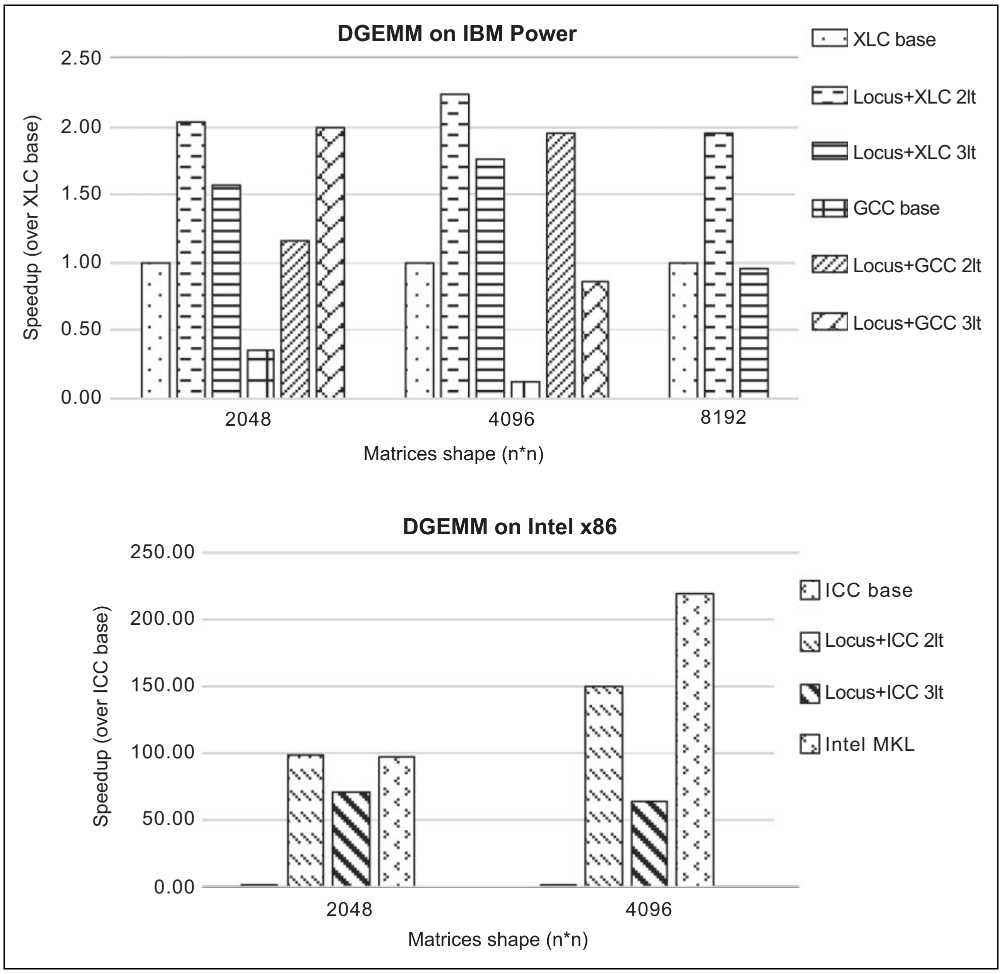

Figure 8. Results for matrix–matrix multiplication on IBM Power 9 and Intel x86. It shows results for the baseline version and the best variant generated by Locus compiled with XLC and GCC. *2lt* refers to two-level hierarchical tiling and *3lt* to three-level hierarchical tiling.

The search time lasted up to 5 h for each of the experiments. While earlier experiments, which only considered two-level tiling, only took on average 1.5 h, the search time has significantly increased after the addition of the third-level tiling to the optimization space. The addition of an third-level tiling within an *OR* block extended the search process because the optimization space in a flattened representation is not aware of the dimensions dependent on other dimensions. In this case, the dimensions `tileI_3`, `tileK_3`, and `tileJ_3` only matter when the second block within the *OR* block is selected; otherwise, their values are ignored in the code generation and do not affect the variant performance. However, the search module, OpenTuner in this case, is not aware of that and may infer that those values are important. In other words, the search module processes an optimization space much bigger that it actually needs to explore and wastes time exploring configurations that do not improve performance.

In Table 3, we show the tiling shapes for the best variants for each matrix shape on the two platforms. The tiling shapes are very different from each other and result in input- and platform-specific variants to be saved in the database.

4.2. Fixed versus random initial configuration

As mentioned above, a Locus program can set the initial configuration of the search process. Figure 9 compares two search strategies for matrix–matrix multiplication on IBM Power. The first search strategy, which we call *fixed*, starts with a configuration provided by the Locus program. This configuration is the one identified as the best for matrix–matrix multiplication on an Intel x86. The second search strategy uses a random configuration selected by the search module.

For the comparison, each of the two forms of search, fixed and random, is run 10 times. For each run, we record the best execution time of matrix–matrix multiplication as the search progresses. The plot shows the best values obtained by each of the 10 runs at different times of the search. The random search in some cases produces slightly better execution times, but it can also produce much slower values even for long searches. We conclude that although the use of a initial configuration based on results from another platform did not guarantee finding the best variant, it confined the search outcome

Table 3. Tiling shapes of the best variant for the matrix–matrix multiplication generated by Locus.[a]

Platform	Matrices Shape	Compiler	tile I	tile K	tile J	tile I2	tile K2	tile J2
IBM Power	2048	XLC	256	128	2048	256	16	8
	4096	XLC	512	512	512	4	512	256
	8192	XLC	8192	4096	8192	8192	64	16
Intel x86	2048	ICC	512	1024	32	512	64	32
	4096	ICC	512	128	2048	256	32	32

[a]Two-level tiling was faster than three-level tiling for all results.

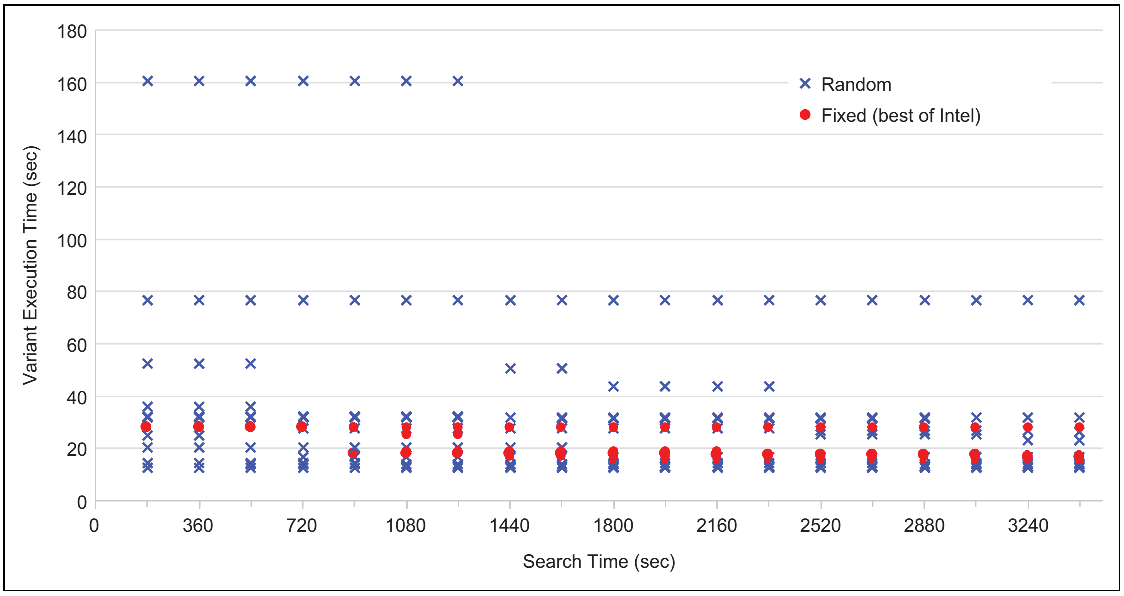

Figure 9. Execution time of best variant found as a function of search time for matrix–matrix multiplication on IBM Power. Comparing random (the default) and fixed initial search configurations. The fixed one is the best found for the Intel x86 on a previous search process.

to a more certain and narrower range (all results are below 28 s).

4.3. 3-D heat stencil

The 3-D heat baseline code is shown in Figure 6. The optimization program in Figure 7 tiles the accesses to the Y dimension of the XYZ input volume. It first stripmines the j loop (responsible for traversing the Y dimension) by calling *RoseLocus.StripMine*. Then, *RoseLocus.Interchange* is used to change the order of the loops by moving the created loop to the outermost position. This tiling increases the reuse of the elements of the XY plane as it traverses the Z dimension and improves performance when the XY plane does not fit on last private level cache. The tiling appears most effective when the problem is large enough that 4 XY (three for the input and one for the output) planes do not fit into cache.

Figures 10 and 11 present weak scaling and strong scaling performance results comparing the baseline version and the best tiled variant generated using Locus. The results were evaluated using 1, 10 (1 for each core), and 20 processes (2 for each core; HyperThreading is available on the processor) on Intel x86; and 1, 20, 40, and 80 processes on IBM Power. The variants and the baseline were compiled with ICC on Intel x86 and XLC on IBM Power. For the concurrent processes evaluation, GNU `parallel` command (Tange, 2011) was used to execute in parallel the same binary.

4.3.1. Weak scaling.

The stencil was executed on a 256^3 mesh of double-precision elements. On the Intel x86 with only one process on the socket, the tiled code does not show any improvement in the aggregated number of millions of stencils performed per second. However, as the access to the cache and memory becomes more competitive with the increasing number of processes, the tiled code performs better. The aggregated performance of the tiled code is approximately 20% higher compared to the baseline when running 10 processes (1 per core). The aggregated performance of the tiled code when using 20 process is slightly worse than with 10 process, which demonstrates the saturation of memory subsystem. It is, however, 50% higher than the baseline performance. Similar conclusions can be drawn from the experiments on the IBM Power.

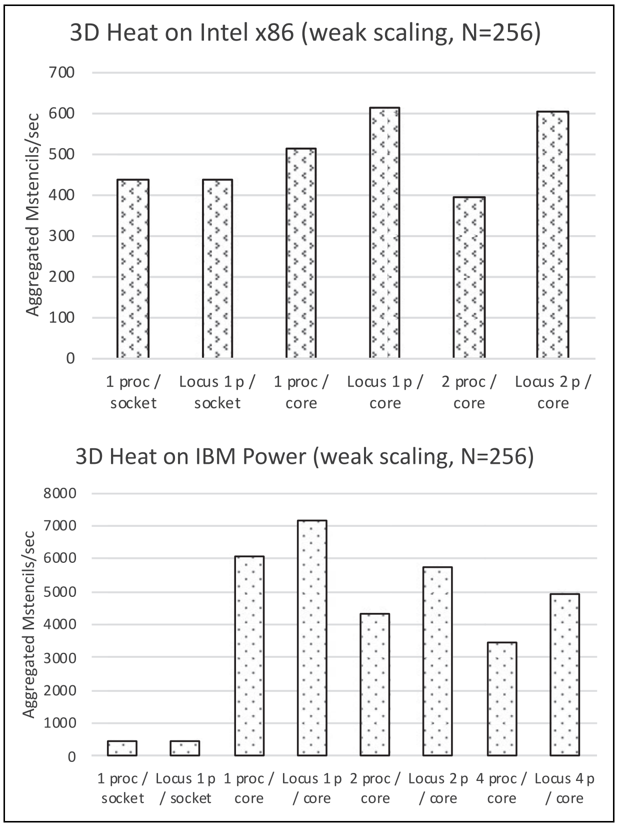

Figure 10. Weak scaling results for the 3-D Heat stencil (256^3 mesh size) on Intel x86 and IBM Power 9. It shows the execution of 1, 10, and 20 processes running concurrently for Intel x86; 1, 20, 40, and 80 running concurrently on IBM Power 9. Tiling appears most effective as the number of processes increases.

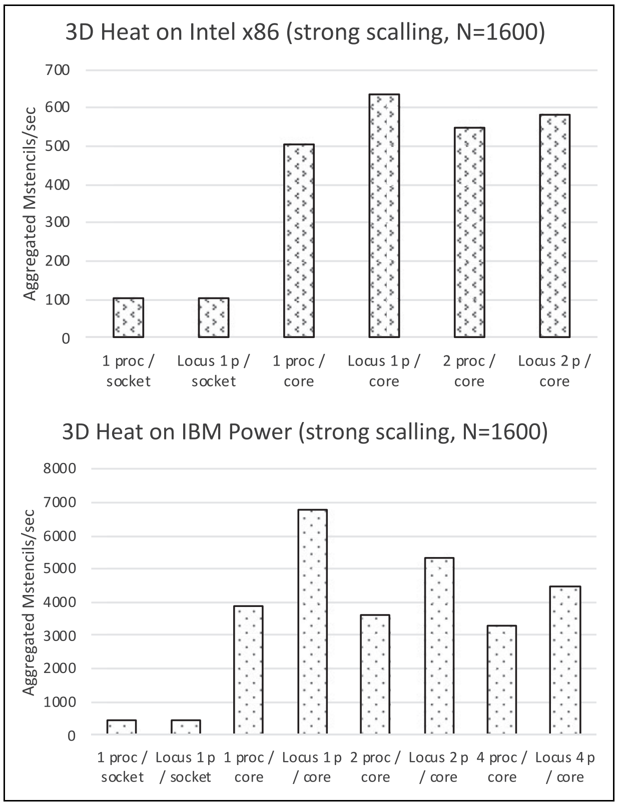

Figure 11. Strong scaling results for the 3-D Heat stencil (1600^3 mesh size) on Intel x86 and IBM Power 9. Only the *Z* dimension of the *XYZ* input volume is split among the processors. It shows the execution of 1, 10, and 20 processes running concurrently for Intel x86; 1, 20, 40, and 80 running concurrently on IBM Power 9. Tiling appears most effective as the number of processes increases.

Despite the processes being independent of each other, the aggregated performance of the system does not follow a linear increase as the number of processes increases. The memory bandwidth is a bottleneck in which optimizations such as tiling can help mitigate its limitations. This contrasts with the observations in Datta et al. (2009), for example, though those were for systems nearly a decade ago, and (with the exception of results on the Cell processors), only one core was used in their work (see Table 2.1 of Datta et al., 2009). However, similar to the findings in that paper and our own more recent work with the XPACC application program, it remains important to make the inner loop use stride-one indexing and be as long as possible, so as to exploit both vectorization and memory optimizations such as prefetch.

The tile values for the best variants decreased as the number of processes increased. On Intel x86, for 10 processes the best found tiling value was 64, and for 20 processes was 16. On IBM Power, for 20 processes the best found tiling value was 64, for 40 was 32, and for 80 was 16. As the caches are shared among more processes, the tile values have to be smaller to accommodate the increasing amount of data.

The optimal variant differed as the concurrency in the processor increased; this information can be added to the database as a tag to the variants. And, in the same way as for the matrix–matrix multiplication, the optimal code is dependent on the problem size.

4.3.2. Strong scaling. Figure 11 presents the results for a mesh of 1600^3 double-precision elements. Through tiling Locus was able to generate faster stencil code, similar to the results from the weak scaling experiments.

The best tiling values found on Intel x86 were 1024 for 1 process, 512 for 10 processes, and 256 for 20 processes. On IBM Power the best tiling values were 32 for 1 process, 16 for 20 processes, 8 for 40 processes, and 4 for 80 processes. Once again, as the number of processes increased, the tile size decreased. The best tiling values found on IBM Power were significantly smaller than the ones found on Intel x86. The gaps between the baseline and best variant were also bigger on IBM Power.

5. Conclusions and future work

In this article, we describe the requirements that we believe are necessary for making automatic performance tuning widely adopted. We present the design and implementation of a system that fulfill these requirements. It includes a domain-specific language that is able to represent complex collections of transformations, an interface to integrate external modules, and a database to manage platform-specific efficient code. The database allows the system users to access optimized code without having to install the code generation toolset. After all, the system presents an approach for performance portability.

We showed two examples that used the system to generate optimized code for two different platforms. Locus was able to generate matrix–matrix multiplication code that outperformed the IBM XLC internal hand-optimized version by 2X on the Power 9 processors. On Intel x86, Locus was able to generate code with performance comparable to Intel MKL's, which is also hand-optimized and platform-specific. The 3-D heat stencil optimized by Locus was up to 75% more efficient in the aggregated performance compared to the baseline version.

We also showed the benefits of using a fixed configuration based on results from a different platform as the search starting point, which confined the search outcome to a narrower range.

The performance attained varied significantly according to the platform and the input dimensions and shows the value of having a database that saves efficient platform-specific code for each code region.

One important goal of the system has been the optimization of the large, complex multi-physics application being developed at the XPACC, 2018. At XPACC, the initial step, and not less challenging, was to define the baseline version out of a code that had already evolved into an optimized version. Subsequently, all the hand-optimized code regions were included in the Locus system as a preferred version. These preferred versions are loaded into the application at build time. We are currently working on using the code transformations available on the system to automatically generate and autotune these preferred versions.

As future work, we plan to evaluate other platforms and the system on more complex applications. We also plan to evaluate other search modules to speedup the search process.

Declaration of Conflicting Interests

The author(s) declared no potential conflicts of interest with respect to the research, authorship, and/or publication of this article.

Funding

The author(s) disclosed receipt of the following financial support for the research, authorship, and/or publication of this article: This material is based in part upon work supported by the Department of Energy, National Nuclear Security Administration, under Award Number DE-NA0002374 and by the National Science Foundation under Award Number 1533912. This work utilizes resources supported by the National Science Foundation's Major Research Instrumentation program, grant #1725729, as well as the University of Illinois at Urbana-Champaign.

ORCID iD

Thiago SFX Teixeira https://orcid.org/0000-0002-8031-0652

Note

1. We used GNU objdump for this analysis.

References

Ansel J, Chan C, Wong YL, et al. (2009) PetaBricks: a language and compiler for algorithmic choice. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Dublin, Ireland, 15–21 June 2009, pp. 38–49. ACM. DOI: 10.1145/1543135.1542481.

Ansel J, Kamil S, Veeramachaneni K, et al. (2014) OpenTuner: an extensible framework for program autotuning. In: *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation, PACT '14*, Edmonton, Alberta, Canada, 24–27 August 2014, pp. 303–316. New York, NY, USA: ACM. ISBN 978-1-4503-2809-8, DOI: 10.1145/2628071.2628092.

Balay S, Abhyankar S, Adams MF, et al. (2018) PETSc web page. Available at: http://www.mcs.anl.gov/petsc (accessed 24 July 2019).

Balay S, Gropp WD, McInnes LC, et al. (1997) Efficient management of parallelism in object oriented numerical software libraries. In: Arge E, Bruaset AM and Langtangen HP (eds) *Modern Software Tools in Scientific Computing*. Basel: Birkhäuser Press, pp. 163–202.

Basili VR, Carver JC, Cruzes D, et al. (2008) Understanding the high-performance-computing community: a software engineer's perspective. *IEEE Software* 25(4): 29–36.

Bergstra J, Yamins D and Cox DD (2013) Making a science of model search: hyperparameter optimization in hundreds of dimensions for vision architectures. In: *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28, ICML'13. JMLR.org*, Atlanta, GA, USA, 16–21 June 2013, pp. I–115–I–123. Available at: http://dl.acm.org/citation.cfm?id=3042817.3042832

Bilmes J, Asanovic K, Chin CW, et al. (1997) Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology. In: *Proceedings of the 11th International Conference on Supercomputing, ICS '97*, Vienna, Austria, 7–11 July 1997, pp. 340–347. New York, NY, USA: ACM. ISBN 0-89791-902-5, DOI:10.1145/263580.263662. Available at: http://doi.acm.org/10.1145/263580.263662

Bondhugula U, Hartono A, Ramanujam J, et al. (2008) A practical automatic polyhedral parallelizer and locality optimizer. In: *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '08*, Tucson, AZ, USA, 7–13 June 2008, pp. 101–113. New York, NY, USA: ACM. ISBN 978-1-59593-860-2, DOI:10.1145/1375581.1375595. Available at: http://doi.acm.org/10.1145/1375581.1375595

Chen C, Chame J and Hall M (2008) *CHiLL: A framework for composing high-level loop transformations*. Technical report, University of Utah.

Chung IH and Hollingsworth JK (2004) Using information from prior runs to improve automated tuning systems. In: *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing, SC '04*, Pittsburgh, PA, USA, 6–12 November 2004, p. 30. Washington, DC, USA: IEEE Computer Society. ISBN 0-7695-2153-3, DOI: 10.1109/SC.2004.65.

Dagum L and Menon R (1998) OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science and Engineering* 5(1): 46–55.

Datta K, Kamil S, Williams S, et al. (2009) Optimization and performance modeling of stencil computations on modern microprocessors. *SIAM Review* 51(1): 129–159.

Donadio S, Brodman J, Roeder T, et al. (2006) A language for the compact representation of multiple program versions. In: *Proceedings of the 18th International Conference on Languages and Compilers for Parallel Computing*, *LCPC'05*, Hawthorne, NY, USA, 20–22 October 2005, pp. 136–151. Berlin, Heidelberg: Springer-Verlag. ISBN 3-540-69329-7, 978-3-540-69329 -1, DOI: 10.1007/978-3-540-69330-7_10.

Edwards HC, Trott CR and Sunderland D (2014) Kokkos: enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing* 74(12): 3202–3216. Available at: http://www.sciencedirect.com/science/article/pii/S0743731514001257 (accessed 24 July 2019). Domain-Specific Languages and High-Level Frameworks for High-Performance Computing.

Fatahalian K, Knight TJ, Houston M, et al. (2006) Sequoia: programming the memory hierarchy. In: *Proceedings of the ACM/IEEE of SC 2006 Conference*, Tampa, FL, USA, 11–17 November 2006, pp. 4–4. DOI: 10.1109/SC.2006.55.

Frigo M (1999) A fast Fourier transform compiler. In: *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation, PLDI '99*, Atlanta, Georgia, USA, pp. 169–180. New York, NY, USA: ACM. ISBN 1-58113-094-5, DOI: 10.1145/301618.301661.

Hartono A, Norris B and Sadayappan P (2009) Annotation-based empirical performance tuning using Orio. In: *2009 IEEE International Symposium on Parallel Distributed Processing*, May 2009, pp. 1–11. DOI: 10.1109/IPDPS.2009.5161004.

Hornung RD and Keasler JA (2014) *The RAJA portability layer: Overview and status*. Technical Report LLNL-TR-661403, Lawrence Livermore National Lab. DOI: 10.2172/1169830.

Keryell R, Ancourt C, Coelho F, et al. (1996) *Pips: A workbench for building interprocedural parallelizers, compilers and optimizers*. Technical Report A/289, Centre de Recherche en Informatique, Ecole des Mines de Paris.

Li X, Garzarán MJ and Padua D (2004) A dynamically tuned sorting library. In: *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, CGO '04, pp. 111. Washington, DC, USA: IEEE Computer Society. ISBN 0-7695-2102-9, Available at: http://dl.acm.org/citation.cfm?id=977395.977663 (accessed 24 July 2019).

Lidman J, Quinlan DJ, Liao C, et al. (2012) ROSE: FTTransform - a source-to-source translation framework for exascale fault-tolerance research. In: *IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN 2012)*, Boston, MA, USA, 25–28 June 2012, pp. 1–6. IEEE. DOI:10.1109/DSNW.2012.6264672.

Muralidharan S, Shantharam M, Hall M, et al. (2014) Nitro: a framework for adaptive code variant tuning. In: *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, Phoenix, AZ, USA, 19–23 May 2014, pp. 501–512. IEEE. DOI: 10.1109/IPDPS.2014.59.

Pennycook SJ, Sewall JD and Lee VW (2016) A metric for performance portability. *arXiv e-prints*. Available at: https://arxiv.org/abs/1611.07409 (accessed 24 July 2019).

Püschel M, Moura JMF, Johnson J, et al. (2005) SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation* 93(2): 232–275.

Ragan-Kelley J, Adams A, Sharlet D, et al. (2017) Halide: decoupling algorithms from schedules for high-performance image processing. *ACM Commun* 61(1): 106–115.

Steuwer M, Fensch C, Lindley S, et al. (2015) Generating performance portable code using rewrite rules: from high-level functional expressions to high-performance OpenCL code. *ACM SIGPLAN Notices* 50(9): 205–217.

Tang Y, Chowdhury RA, Kuszmaul BC, et al. (2011) The Pochoir stencil compiler. In: *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures SPAA '11*, San Jose, California, USA, pp. 117–128. New York, NY, USA: ACM. ISBN 978-1-4503-0743-7, DOI: 10.1145/1989493.1989508.

Tange O (2011) GNU parallel - the command-line power tool. *login: The USENIX Magazine* 36(1): 42–47. Available at: http://www.gnu.org/s/parallel (accessed 24 July 2019).

Teixeira TSFX, Ancourt C, Padua D, et al. (2019) Locus: a system and a language for program optimization. In: *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization CGO 2019*, Washington, DC, USA, pp. 217–228. Piscataway, NJ, USA: IEEE Press.

Tiwari A, Hollingsworth JK, Chen C, et al. (2011) Auto-tuning full applications: a case study. *International Journal of High Performance Computing Applications* 25(3): 286–294.

Vuduc R, Demmel JW and Yelick KA (2005) OSKI: a library of automatically tuned sparse matrix kernels. *Journal of Physics: Conference Series* 16(1): 521. Availble at: http://stacks.iop.org/1742-6596/16/i=1/a=071

Whaley RC and Dongarra JJ (1998) Automatically tuned linear algebra software. In: *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing, SC '98*, pp. 1–27. Washington, DC, USA: IEEE Computer Society. ISBN 0-89791-984-X. Available at: http://dl.acm.org/citation.cfm?id=509058.509096 (accessed 24 July 2019).

Wolfe M (2016) Compilers and more: what makes performance portable? Available at: https://www.hpcwire.com/2016/04/19/compilers-makes-performance-portabl (accessed 19 April 2016).

XPACC (2018) Center for the exascale simulation of plasma-coupled combustion web page. Available at: http://xpacc.illinois.edu; http://xpacc.illinois.edu (accessed 24 July 2019).

Yi Q, Seymour K, You H, et al. (2007) POET: parameterized optimizations for empirical tuning. In: *2007 IEEE International Parallel and Distributed Processing Symposium*, Rome, Italy, 26–30 March 2007, pp. 1–8. IEEE. DOI: 10.1109/IPDPS.2007.370637.

Author biographies

Thiago SFX Teixeira is a PhD candidate in computer science at the University of Illinois at Urbana-Champaign. Prior to joining the University of Illinois, Thiago worked for 6 years in the research and development of parallel scientific applications for the oil and gas industry. Thiago holds BS and MS in computer science from the Federal University of Minas Gerais, Brazil. His master's dissertation was awarded as the best dissertation in computer architecture and high performance computing by the Brazilian Computer Society in 2011.

William Gropp is the director and chief scientist of the National Center for Supercomputing Applications and holds the Thomas M. Siebel Chair in the Department of Computer Science at the University of Illinois in Urbana-Champaign. He received his PhD in computer science from Stanford University in 1982. He was on the faculty of the Computer Science Department of Yale University from 1982 to 1990 and from 1990 to 2007, he was a member of the Mathematics and Computer Science Division at Argonne National Laboratory. His research interests are in parallel computing, software for scientific computing, and numerical methods for partial differential equations. He is a fellow of AAAS, ACM, IEEE, and SIAM and a member of the National Academy of Engineering.

David Padua is Donald Biggar Willet Professor in engineering in the University of Illinois at Urbana-Champaign. He has served as program committee member, program chair, or general chair to more than 70 conferences and workshops. He was the editor-in-chief of Springer-Verlag's *Encyclopedia of Parallel Computing* and is currently a member of the editorial board of the *IEEE Transactions of Parallel and Distributed Systems*, the *Journal of Parallel and Distributed Computing*, and the *International Journal of Parallel Programming*. He has supervised the dissertations of 35 PhD students. He has published more than 170 papers in programming languages, compilers, tools, and parallel machine design. He was awarded the 2015 IEEE Computer Society Harry H. Goode Award. In 2017, he received an honorary doctorate from the University of Valladolid in Spain. He is a fellow of the ACM and the IEEE.