Locus: A System and a Language for Program Optimization

Thiago S. F. X. Teixeira*, Corinne Ancourt†, David Padua*, William Gropp*

*Department of Computer Science
University of Illinois at Urbana-Champaign, USA
{tteixei2,padua,wgropp}@illinois.edu

†MINES ParisTech, PSL University, France
corinne.ancourt@mines-paristech.fr

***Abstract*—We discuss the design and the implementation of Locus, a system and a language to orchestrate the optimization of applications. The increasing complexity of machines and the large space of program variants, produced by the many transformations available, conspire to make compilers deliver unsatisfactory performance. As a result, optimization experts must intervene to manually explore the space of program variants seeking the best version for each target machine. This intervention is unproductive, and maintaining and managing sequences of transformations as new architectures are adopted and new application features are incorporated is challenging.**

Locus allows collections of program transformation sequences to be specified separately from the application code. The language is able to represent in a clear notation complex collections of transformations that are applied to code regions selected by the programmer. The system integrates multiple optimization modules as well as search modules that facilitate the efficient traversal of the space of program variants. Locus is intended to help experts in the optimization process, specially for complex, long-lived applications that are to be executed on different environments. Four examples are presented to illustrate the power and simplicity of the language. Although not the primary focus of this paper, the examples also show that exploring the space of variants typically leads to better performing codes than those produced by conventional compiler optimizations that are based on heuristics.

***Index Terms*—code generation, optimization, compilers, domain-specific language.**

I. Introduction

Due to the complexity of today's machines, the gap between the performance of hand-tuned and compiler-generated code has grown substantially and software developers must devote significant time to benefit from computing power. Furthermore, each architecture typically requires a different sequence of optimizations to attain a high fraction of its nominal peak speed. This complicates performance portability and code maintainability.

A critical challenge in developing complex, long-lived applications is to figure out how to manage different optimized versions of the same code tailored to different architectures and keep them up to date as new features are added to the application.

Although programmers can make use of a catalog of program optimizations aimed at locality improvement, enhancement of instruction level parallelism, latency hiding, improved register utilization, and vectorization [1]–[3], their implementation is not straightforward, and composing and implementing a collection of transformations correctly into a final optimal version further increase the difficulty. In addition, applying optimizations tends to complicate the code, hurt readability, and the result is unlikely to be performance-portable.

Existing tools commonly require considerable manual refactoring to improve performance [4]–[6], and provide little control over the steps being carried out. They are also not prepared to coexist with other tools and cannot be incrementally adopted.

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). It combines expert knowledge with empirical search, automates much of the optimization process and gives total control to the developers.

Locus decouples the performance expert role from the application expert role (separation of concerns). It allows the use of architecture-specific optimizations while keeping the code maintainable in the long term. Locus orchestrates the application of transformations to a *baseline version* of the code and coordinates the empirical search for the best sequence of optimizations and their parameters. The application of the optimizations and the empirical search are programmed using a domain-specific language (DSL) in what we refer as *optimization programming*. An optimization program can make use of transformation modules from different collections available in the system, such as the ones developed based on Rose [7] and Pips [8].

The baseline version is defined by the developer, but it should be as readable as possible and avoid architecture- or compiler-specific optimizations.

Locus makes use of a powerful language for programming the orchestration of optimizations and for representing complex optimization spaces. It is designed to facilitate the integration of different optimization and empirical search modules.

978-1-7281-1436-1/19/$31.00 © 2019 IEEE

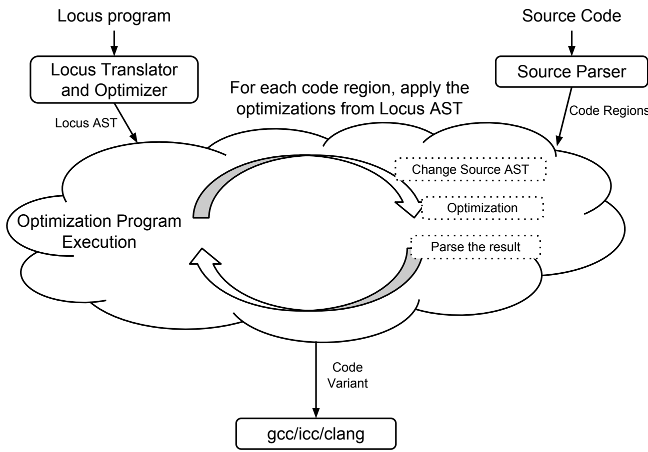

Fig. 1. The Locus system.

The main contributions of this work are:

- Locus, which is, as far as we know, the first programming language capable of representing complex spaces of program variants generated by multiple transformation sequences for mainstream programming languages (C, C++, Fortran). It is similar to what other systems have done through rewriting rules [9], [10] and code generation [11], but the manipulation of mainstream languages requires a different approach to enable the application of compiler transformations and the concise representation of collections of transformation sequences;
- a translator and an optimizer for Locus. Using a translator, instead of a library, makes the notation less verbose and enables the automatic optimization of Locus programs;
- the first system that is able to manage the application of different transformations to different code regions;
- and a system that brings together transformations, optimization space generation, and traversal of the search space.

Section II provides a detailed description of the Locus system. Section III describes the optimization language, whereas Section IV presents the strategy used for integrating external transformation and search modules. Section V presents experimental results. We finish with a discussion of related work in Section VI, and conclusions and future work in Section VII.

II. Locus Optimization System

Program code is typically altered by the numerous optimizations needed for each possible target environment (architecture and software stack). Over time, the code becomes unrecognizable, difficult to maintain, and challenging to modify. Moreover, as the program evolves, it is difficult to keep the optimizations up to date. In an attempt to ease this problem we present the design and implementation of Locus, a system for optimizing complex, long-lived applications for efficient execution on different environments.

The system is based on the idea that optimizations should not be embedded in the application code. Instead, a program written in a DSL specifies the optimization sequences to be explored, and is kept separate from the source code to be optimized. In the source code, regions of interest (also referred as *code regions*) are marked and given an identifier. The optimization program uses this identifier to specify where to apply each transformation. When multiple regions have the same identifier, the same sequence of optimizations will be applied to each of them.

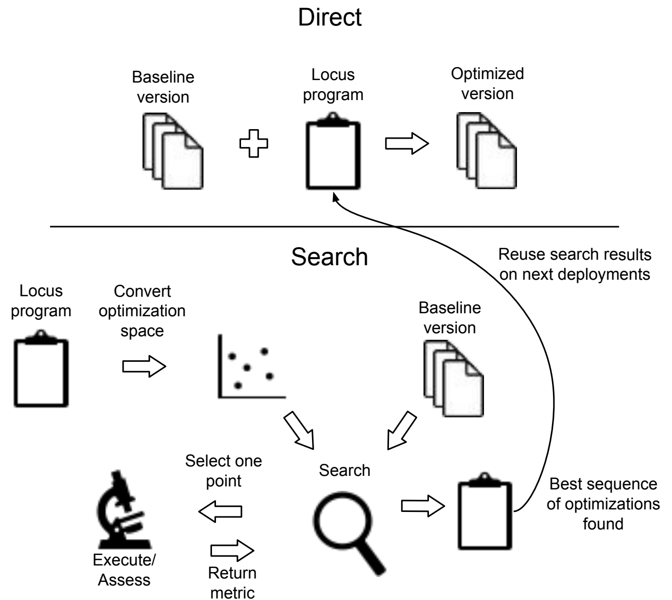

Fig. 2. Two workflows: *direct* and *search*.

The system is non-prescriptive, which means that if none of the transformations in the optimization program can be applied or improve performance, the baseline version (original code) is used instead. This guarantees that there is always a version available for execution on all targeted machines, even though it might be far from the the fastest possible.

Through the Locus program, the software developer has complete control over the optimization sequences that are to be attempted to improve performance. It can be said that a Locus program defines an optimization space that can be very large and in many cases impossible to be fully traversed. Therefore, the system is designed to use methods that can find efficient solutions without traversing the whole space. Locus programs can be developed separately from the base code, enabling separation of concerns.

The system itself does not check for correctness. It is left to each module to check (or not) whether the optimization is legal. Given the limitations of the legality check implemented by some tools (especially with pointers), a programmer might feel interested in enforcing an optimization when she/he knows it is legal.

A prototype of Locus has been implemented in Python. It accepts C, C++ and Fortran programs. The system (Figure 1) contains two classes of front-ends that read, parse, and analyze

their respective inputs: one for the optimization language and the other for the baseline source code.

The current prototype is limited to source-to-source optimizations and cannot handle lower level optimizations (e.g., register allocation, instruction reordering). At the source level, it is easier to mark and keep track of optimizations intended for each code region. Previous work [12] has required manual intervention to identify tagged code in lower-level representations. The system's approach, though, is powerful enough to apply any compiler optimization available to the system as long as the subsequent optimization comprehend the previous optimization output.

The Locus language is dynamically typed and the system includes a translator and an optimizer for it. At run time, the resulting high-level representation of the optimization program is interpreted to generate variants of the source code.

Besides the language, 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 used and integrated in a single system. This also enables comparison of modules, and once they are integrated in the system, it becomes easy to empirically search for the best result across modules.

The implementation of a wrapper function that extends the interface is required to enable the communication between modules and system. The wrapper function is responsible for handling and returning to the system the exit status (e.g., successful, error, illegal) of a module's invocation. In the case of a search module, it is also necessary to implement a function that converts between search space representations.

The programmer defines code regions in the source code using C pragmas or Fortran comments. There are two types of annotations: block and loop. Figure 3 is an example of code region with a loop annotation.

```
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;
}
```

Fig. 3. A C program containing a loop nest with the identifier to be referenced in the optimization program.

The block annotation indicates the begin and the end of the code region. The loop annotation applies to the first loop nest after the annotation. Loop annotations are for loop transformations, and block annotations for alternative algorithm selections or optimizations comprising multiple code regions.

In Locus, it is possible to specify sequences of transformations for a particular block or loop or for a collection of these (when multiple regions are labeled with the same identifier). This ability to target transformations to specific regions is useful, because certain transformations are only effective on particular code segments. For example, tiling is effective only on loops that access data items multiple times, and distribution for loops enclosing multiple statements. Besides, some transformation modules only work on particular classes of code segments although others are more generally applicable. For example, a loop interchange module may only work on perfectly nested loops, whereas others can operate on any kind of loop nest.

Figure 2 presents the two workflows of Locus: *direct* and *search*. The *direct* workflow applies one sequence of transformations and generates an optimized variant of the baseline version. The *search* workflow applies when the Locus program contains search constructs. It starts by converting the optimization space from the Locus representation to the search module's representation. The search module returns a point from the optimization space that contains a single value for each search construct. These values are used to build a representation of the optimization program that is applied to the baseline. The resulting code is evaluated according to a metric (e.g., execution time). The metric is then returned to the search module which may use that to decide which point in the optimization space to assess next. The number of assessments made by the search module is an important parameter. The more assessments are made the higher the chance of finding the optimization sequence closest to the optimal. At the end, the result is a Locus direct program that can be shipped with the baseline source code to be reused for machines with similar environments.

Changing the source code may render the optimizations defined in a Locus program illegal, incorrect or useless. It is necessary to keep the coherence between the code regions defined in the source code and the optimizations that are supposed to be applied to them. The solution we found is to hash the code region and use the key to check for code changes to be able to warn the programmer.

A. Optimization Space

The optimization space specified by a Locus program is defined by a number of dimensions including compilers, their versions and flags, data structures, loop transformations, and their parameters (e.g., tile sizes, unroll factors).

Conditional expressions can be used to segment and represent the optimization space as a decision tree. Optimizations on conditional spaces have been shown on selecting hyperparameters to correctly apply machine learning algorithms [13]. Representing and efficiently traversing conditional spaces is more challenging than working with flattened spaces and not many techniques are available to manipulate conditional spaces.

The use of conditional spaces fits well on the code optimization process. In this domain, *conditionality* can be seen, for instance, in the data structures selection process, which

heavily depends on the target architecture. In the same way, the compiler flags depend on the compiler used as they significantly vary across compilers. Numerous choices depend on the architecture used, including the compilers, data structures, and loop transformations. The loop transformations commonly depend on how the data are laid out as well as on memory hierarchy.

The programmer exposes the optimization space using the Locus constructs and each code variant becomes a point in that space.

```
⟨coderefreg⟩ ::= ‘CodeReg’ NAME ⟨block⟩
⟨optseqdef⟩ ::= ‘OptSeq’ NAME ‘(’[⟨arglist⟩]‘)’ ⟨block⟩
⟨tooldef⟩ ::= ‘Module’ NAME ⟨block⟩
⟨querydef⟩ ::= ‘Query’ NAME ‘(’[⟨arglist⟩]‘)’ ⟨block⟩
⟨searchblock⟩ ::= ‘Search’ ⟨block⟩
⟨extern⟩ ::= ‘extern’ ⟨mol⟩ ‘;’
⟨import⟩ ::= ‘import’ STRING ‘;’
⟨defmethod⟩ ::= ‘def’ NAME ‘(’[⟨arglist⟩]‘)’ ⟨block⟩
⟨block⟩ ::= ‘{’ ⟨subblock⟩ ‘}’
⟨subblock⟩ ::= ⟨block⟩ ( ‘OR’ ⟨block⟩ )+ | ⟨block⟩ | ⟨stmt⟩
⟨stmt⟩ ::= ⟨setstmt⟩ | ⟨compoundstmt⟩
⟨setstmt⟩ ::= ⟨smallstmt⟩ ( ‘;’ ⟨smallstmt⟩ )* ‘;’
⟨compoundstmt⟩ ::= ⟨ifstmt⟩ | ⟨forstmt⟩ | ⟨whilestmt⟩
⟨smallstmt⟩ ::= ⟨optionalstmt⟩ | ⟨assignstmt⟩
⟨assignstmt⟩ ::= ⟨testlist⟩ ‘=’ ⟨optionalstmt⟩
⟨optionalstmt⟩ ::= ‘*’⟨orexprstmt⟩ | ⟨orexprstmt⟩
⟨orexprstmt⟩ ::= ⟨testlist⟩ (‘OR’ ⟨testlist⟩)*
⟨forstmt⟩ ::= ‘for’ ‘(’ ⟨smallstmt⟩ ‘;’ ⟨test⟩ ‘;’ ⟨smallstmt⟩ ‘)’ ⟨block⟩
⟨whilestmt⟩ ::= ‘while’ ⟨test⟩ ⟨block⟩
⟨ifstmt⟩ ::= ‘if’ ⟨test⟩ ⟨block⟩ (‘elif’ ⟨test⟩ ⟨block⟩)* [‘else’ ⟨block⟩]
⟨rangeexpr⟩ ::= ⟨expr⟩ .. ⟨expr⟩ [.. ⟨expr⟩]
⟨testlist⟩ ::= ⟨test⟩ (‘,’ ⟨test⟩)* [‘,’]
⟨test⟩ ::= ⟨andtest⟩ (‘||’ ⟨andtest⟩)*
⟨andtest⟩ ::= ⟨nottest⟩ (‘&&’ ⟨nottest⟩)*
⟨nottest⟩ ::= ‘not’ ⟨nottest⟩ | ⟨comparison⟩
⟨comparison⟩ ::= ⟨expr⟩ (⟨compop⟩ ⟨expr⟩)*
⟨expr⟩ ::= ⟨term⟩ | ⟨expr⟩ (‘+’|‘−’) ⟨term⟩
⟨term⟩ ::= ⟨power⟩ | ⟨term⟩ (‘*’|‘/’|‘%’) ⟨power⟩
⟨power⟩ ::= ⟨mol⟩ [‘**’ factor]
⟨mol⟩ ::= ⟨mol⟩ ‘(’ [⟨arglist⟩] ‘)’ | ⟨mol⟩ ‘[’[sublist]‘]’
    | ‘enum’ ‘(’ testlist ‘)’
    | ‘poweroftwo’ ‘(’ ⟨rangeexpr⟩ ‘)’
    | ‘integer‘ ‘(’ ⟨rangeexpr⟩ ‘)’
    | ‘permutation’ ‘(’ (⟨test⟩ | ⟨listmaker⟩) ‘)’
    | ⟨mol⟩ ‘.’ NAME | ⟨atom⟩
⟨atom⟩ ::= NAME | NUMBER | STRING
⟨listmaker⟩ ::= ⟨test⟩ (‘,’ ⟨test⟩ )* [‘,’]
⟨sublist⟩ ::= test (‘,’ test)* [‘,’]
⟨arglist⟩ ::= (⟨argument⟩ ‘,’)* (⟨argument⟩ [‘,’])
⟨argument⟩ ::= ⟨test⟩ | ⟨test⟩ ‘=’ ⟨test⟩
⟨compop⟩ ::= ‘>’ | ‘<’ | ‘==’ | ‘>=’ | ‘<=’ | ‘!=’
```

Fig. 4. The extended Backus-Naur form of Locus.

III. Locus Optimization Language

The optimization program uses a language to define the sequence of optimizations. Figure 4 shows the (partial) extended Backus-Naur form of the language. The heading *CodeReg NAME* indicates that the optimization steps in the *<block>* that follows are to be applied to the regions labeled *NAME*.

To name optimization sequences that are not tied to any code region there is the construct *OptSeq NAME*, which defines a collection of transformations that can be invoked using *NAME*.

```
import "RoseLocus";
def printstatus(type) {
  print "Tiling␣selected:␣"+type;
}
OptSeq Tiling2D() {
  tileI = poweroftwo(2..32);
  tileJ = poweroftwo(2..32);
  RoseLocus.Tiling(loop="0", factor=[tileI,tileJ]);
  return "2D";
}
OptSeq Tiling3D() {
  RoseLocus.Tiling(loop="0", factor=[4,4,8]);
  return "3D";
}
CodeReg matmul {
  tiledim = 4;
  tiletype = Tiling2D() OR Tiling3D();
  printstatus(tiletype);
  if (tiletype == "2D") {
    RoseLocus.Unroll(loop=innermost, factor=tiledim);
  }
}
```

Fig. 5. A Locus program example.

These collections of transformations can be reused and invoked from different *CodeReg*s and other *OptSeq*s. Only blocks within *OptSeq* or *CodeReg* can invoke transformation modules that will operate on the regions of interest. Regular methods can also be implemented using the keyword *def* but they cannot contain any optimization or query call.

Figure 5 is an example of Locus program that tiles a baseline version of matrix-matrix multiplication (Figure 3). It shows a search space that includes two *OptSeq*s: one, `Tiling2D`, for tiling the two outermost loops and another, `Tiling3D`, to tile the three outermost loops, which, in this case, is the whole loop nest. The transformation sequence that applies to the loop labeled `matmul` in Figure 3 has header `CodeReg matmul`. This sequence specifies, using the Locus `OR` operation, that `Tiling2D` and `Tiling3D` will be used to create different collections of points in the space of transformations. `Tiling2D` will create 25 points on the space for the tiles of dimension 2 by 2, 2 by 4, ..., 2 by 32, ... 32 by 32, whereas `Tiling3D` will create one additional point on the search space for a tile of dimension 4 by 4 by 8. When `Tiling2D` is applied, the innermost loop is also unrolled after tiling. The tiling factor parameters in `Tiling2D` are a range of values, whereas the ones for the `Tiling3D` are fixed.

There are also two constructs on the language to represent modules and queries. The *Query* represents procedures to analyze and extract information from a code region. It can only be invoked from inside a *CodeReg* or an *OptSeq*. The *Module* construct represents a set of transformations or queries. The *Query* construct is necessary because, differently than *OptSeq*, its results can be used by search constructs, and, as better explained later in the text, they need their information defined before the search process starts. The rest of this section provides more information about the language.

Search Constructs: Locus has multiple *search constructs* to expose and define the optimization space. The search constructs are:

1) OR Blocks;

2) OR Statements;
3) Optional Statements;
4) *enum*, *integer*, *float*, *permutation*, *poweroftwo*, *loginteger*, and *logfloat* data types.

An *OR* can be used between blocks of code (set of statements inside curly brackets (e.g., *{optA;} OR {optB;}*)) and between statements (e.g., *transfA OR transfB;*). They are used to describe alternative optimization sequences. The statements or blocks are selected at run time and it is not guaranteed that all possibilities will be evaluated; this decision depends on the search module.

In the same way, Locus may have *optional* statements. Any statement (including *OR* statements) marked with a preceding $*$ may or may not be executed and adds a dimension to the space of optimizations. The *OR* statements have precedence over the *optional* construct. The semantics of an optional statement is the same as having an *OR* statement in which one of the options is *None*. To keep this semantics, assignments cannot be optional statements.

The constructs *enum*(<*value*>,...), *permutation*(<*value*>,...), and [*integer*, *float*, *permutation*, *poweroftwo*, *loginteger*, *logfloat*] (<*min*>..<*max*>) are useful for defining collections of values in the search space. The *enum* exposes to the search all values received; it can be used for comparing the performances of different compiler flags (e.g., *enum*(-O2,-O3)). The *permutation* exposes to the search all the permutations of a list. One use of *permutation* is to assess all orders of a loop nest as a parameter of the loop interchange optimization. The *poweroftwo* exposes to the search all values that are power of two on the given range. The *integer* exposes to the search all the integer values in the range. Note that ".." is used to represent a range. The *integer* can be used to find the best dimensions when tiling a loop nest, and *poweroftwo* can reduce the search time as it covers fewer values for the same range. The same reasoning is applied to the *loginteger* and *logfloat* constructs.

The values used in these constructs may also be another search variable. This is specially useful for defining constraints that depend on previously selected variables. For instance, in a sequence that tiles and then unrolls, the unroll factor is limited by the number of loop iterations, which in turn is defined by the block size selected using another search variable. If the block size was selected as 32, the search space for the unroll factor should be constrained by that value. The use of constraints reduces the size of the space and potentially speeds up the search process.

Data Structures: Locus also accepts lists, tuples and dictionaries (in similar fashion to Python). The lists are represented by enclosing elements within square brackets, tuples by enclosing elements within parenthesis, and dictionaries through the construct *dict*. Lists and dictionaries are mutable and tuples are immutable.

Types: Beyond the data structure types (lists and dictionaries) there are two other basic types: numbers and strings. As with tuples, these types are immutable. Strings are surrounded by double quotes. Signed integers and floating-point real values are the types of numbers accepted.

Control Flow: The control flow statements available are *if*, *for*, and *while*. They can be used to decide, during run time, how to proceed the optimization according to previous decisions. For instance, inside *CodeReg* it is possible to define different sequences of optimizations depending on the compiler used. The compiler to be used can be expressed as a variable (e.g., compiler=*enum*("gcc","icc")), and an *if* can be used to execute different sequences of optimizations according to the value assigned to the variable compiler during the search process.

Scope: Each block of code has its own scope; the scope of the control flow constructs, however, is the same as their parent block. Variables defined inside *if*, *for*, and *while* are then possible to be accessed after the end of their execution.

Import: It is possible to import optimization sequences created by others. This is an important feature as experts can share their recipes for common code regions running on similar architectures. It is also used to import modules and optimization definitions to be used on *CodeReg* and *OptSeq*.

Search Block: The *Search* block is used to give the system commands on how to build, run, and measure the chosen performance metrics. The statements in the search block may include flow statements and take actions based on variable selections made in the global scope.

Hierarchical Indexing: This indexing is made of numbers separated by periods. Each number represents a level, starting from 0, in a statement block or a loop nest. The value of the number represents the order of the statement or loop in that level. For instance, "0.0.0" refers to the innermost loop in the Figure 3. In case of two innermost loops, the second one would be referenced as "0.0.1". Using this indexing we are able to reference any statement or loop in a code region.

IV. Modules Integration

An important goal to the system is to have a collaborative environment where external transformation and search modules can be integrated. The following subsections provide more information about the integration of such modules.

A. Integration of Transformation Modules

One significant challenge in developing the system was to integrate and make possible the interaction with different and unrelated optimization modules. The integration programmer is responsible for implementing the translation of the information so that it can be fed from one module to another.

The current implementation deals only with source-to-source transformations. A very common workflow for the integrated modules is: *mark the code region according to the module specification*, *unparse the code (generates source code)*, *apply the optimization*, *parse the code again and translate it back into Locus internal representation for source codes (abstract syntax tree and code region data structure)*.

Several transformations were integrated using this sequence. A wrapper function for each translator implements it using the system interface. Before calling the optimization module, the abstract syntax tree is modified to mark the code region that the module is supposed to optimize.

These marks (e.g., labels, pragmas) depend on the module integrated. After the optimization has finished, the code regions must be identified again as the result is parsed back into the internal abstract syntax tree. The interface is comprised of operations to modify the internal abstract syntax tree, such as: `replaceCoregWithPragma()`, `replaceCoregWithLabel()`, `addPragma()`, `addLabel()`, `replaceLabelWithCoreg()`, `removeLabel()`, `replacePragmaWithCoreg()`, and `removePragma()`.

There are four collections of transformation modules currently available: Pips, RoseLocus, Pragmas, and BuiltIn. Next, we discuss how they interface with the system, and how to use their optimizations.

1) Pips: A source-to-source compilation framework for analyzing and transforming C and Fortran 77 programs [8], Pips has a Python interface and the loop optimizations are invoked on loops marked with labels. Locus adds to the code region a label and removes the pragma that originally identified the region using the `replaceCoregWithLabel()` operation. The changed AST is unparsed in a temporary file; Pips is invoked; after finishing the optimization, the resulting code is read and parsed, and its AST and code region data structure are rebuilt.

The optimizations from Pips available in Locus are: loop unrolling, loop *GenericTiling*, loop fusion, and unroll-and-jam. The *GenericTiling* accepts a matrix representing the tiling transformation, but also two extra parameters to enhance parallelism of the generated code: *tile direction*, which specifies the scanning directions of the tiles, and *local tile direction*, which specifies the scanning directions of the iterations inside each tile.

2) RoseLocus: We implemented our own set of annotation-based source-to-source loop transformations using the Rose infrastructure [7]. Loop unrolling, tiling, interchange, unroll-and-jam, loop invariant code motion, and scalar replacement are available. The interface with RoseLocus is similar to that with Pips. However, instead of labels, pragmas are added to the code regions.

We have also implemented a query to check whether the available modules can compute the data dependences of the source code. The dependences are used by some optimizations to check if the transformation would be legal.

3) Pragmas: Compiler-specific pragmas can be added using this module. In the experiments, `ivdep` and `vector always` are used for enhancing vectorization with the ICC compiler. It is also possible to execute loops in parallel by adding `omp parallel for` pragmas [14]. The module to apply the OpenMP pragma accepts the schedule (dynamic or static) and the chunk size as parameters.

4) BuiltIn: It contains modules to analyze or transform the source code by manipulating the internal AST. Queries to get information about loop nests are available. *ListInnerLoops* and *ListOuterLoops* return a list with the innermost and outermost loops, respectively, from a code region. *IsPerfectLoopNest* returns whether the loop is perfectly nested. And *LoopNestDepth* returns the depth of a loop nest.

It also contains *Altdesc* that is used to replace the code region with external code snippets. Its functionality is similar to having macros in the program. It is mostly used to incorporate hand-optimized kernels into an optimization sequence.

B. Integration of Search Modules

Three functions must be implemented to enable the integration of a search module: 1) *convertOptUniverse*, which is responsible for automatically converting the Locus optimization space of all code regions to the module's search space. The space conversion usually includes converting data types, and defining parameters and options that will be used during the search process; 2) *search*, which is responsible for starting the search process. The search can only start after the conversion is finished; 3) a conversion back to the Locus representation that is used after a point in the space is chosen. This point information contains a selected single value for each search construct on the space. The Locus code can now be interpreted and the optimizations carried out.

Two modules were integrated to explore the space of optimizations: Opentuner [15] and Hyperopt [16]. Next, more details about each tool integration are provided.

1) OpenTuner: The search constructs for *OR* blocks, *OR* statements, and *enum* are converted to OpenTuner's *EnumParameter*. The optional statements are represented using the *BooleanParameter*. The other search constructs have straightforward representations between the two spaces. For instance, *permutation* is represented by *PermutationParameter*.

The use of numerical search variables that depend on other numerical search variables is not natively supported by OpenTuner. Locus supports it, but, to precisely define the optimization space, the minimum and maximum possible values that reach the search variable parameters must be computed. A data-flow analysis through the use-def chains of the Locus program is performed to get the boundary values.

When a point is selected for empirical evaluation and the values of the search variables are known, it is necessary to check whether the dependent variable is valid. For instance, in Figure 7 the second level of tiling (`tileI_2`, `tileK_2`, and `tileJ_2`) depends on the results of the first level (`tileI`, `tileK`, and `tileJ`). In other words, we need to check that the value selected for `tileI_2` is smaller or equal to the one selected for `tileI`; otherwise, this variant is invalidated, and the search process moves on to the next one.

2) HyperOpt: All the non-numerical search constructs are represented by the `choice` statement. The numerical search constructs use `randint` parameter. Hyperopt accepts the use of conditional search space. The use of search variable parameters dependent on other numerical search variables are treated in the same way as in OpenTuner.

C. Optimizations to Locus Programs

Optimizations are applied to the Locus programs to reduce the system's execution time. These optimizations can have a major effect on the search. In the search workflow, the Locus direct program is interpreted for each variant evaluated.

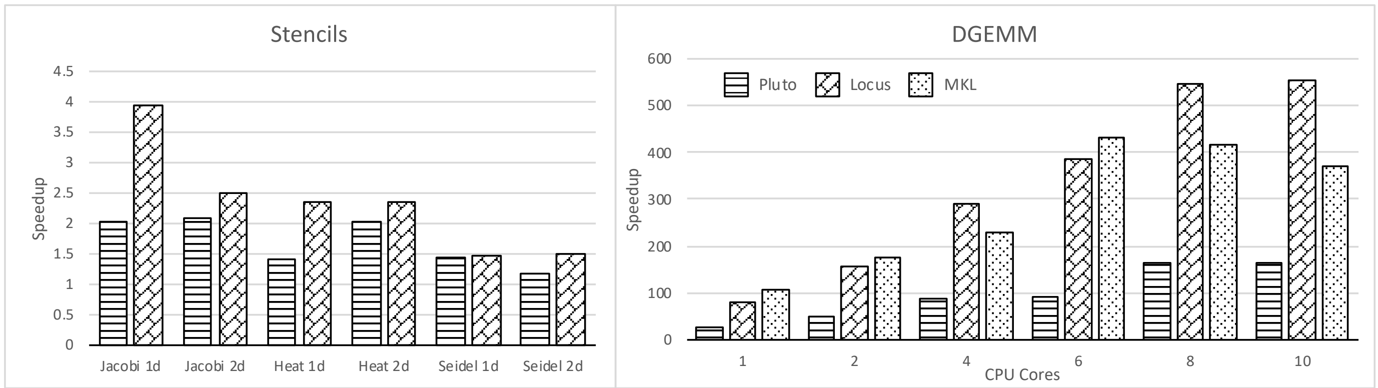

Fig. 6. Speedup for Locus, Pluto, and Intel MKL on Stencils and Matrix-Matrix Multiplication (DGEMM).

Constant propagation, constant folding and dead code elimination are applied before the Locus space is converted to the search module's space, but after the execution of any *Query* operation that is used by any search construct. The *Query* operations are assumed to have a deterministic result throughout the search process if they are used by any search construct. The parameters of the search constructs need to be known when the search is defined. Therefore, these *Query* operations are executed and their values replace their calls on the code.

The use of these optimizations can drastically reduce the search time by reducing the space and improving the quality of variants suggested to be assessed. For instance, in Section V-D, there is one example in which optimizations can only happen if the loop nest depth is greater than 1. Therefore, for loop nests with depth equal to 1, we can exclude from the search space all the search constructs on the code conditional to the loop nest depth greater than 1.

```
Search {
  buildcmd = "make␣clean;␣make";
  runcmd = "./matmul";
}
CodeReg matmul {
  RoseLocus.Interchange(order=[0,2,1]);
  tileI = poweroftwo(2..512);
  tileK = poweroftwo(2..512);
  tileJ = poweroftwo(2..512);
  Pips.Tiling(loop="0", factor=[tileI, tileK, tileJ]);
  tileI_2 = poweroftwo(2..tileI);
  tileK_2 = poweroftwo(2..tileK);
  tileJ_2 = poweroftwo(2..tileJ);
  Pips.Tiling(loop="0.0.0.0",
              factor=[tileI_2, tileK_2, tileJ_2]);
  {
    Pragma.OMPFor(loop="0");
  } OR {
    Pragma.OMPFor(loop="0",
                  schedule=enum("static","dynamic"),
                  chunk=integer(1..32));
  }
}
```

Fig. 7. Locus program for optimizing Matrix-Matrix Multiplication (DGEMM).

V. Experimental Results

In this section, we evaluate the performance of the code generated by Locus. We compare it to the code generated by Pluto [17] (*pet* branch version 0.11.4) with no parameter tuning. The results were obtained on a 10-core Intel Xeon E5-2660 v3 processor clocked at 2.60 GHz (32 KB private L1 instruction cache, 32 KB private L1 data cache, 256 KB private L2 cache, 25 MB shared L3 cache) with 62 GB of RAM, and running Linux kernel version 4.4.0 (x86-64). The compiler used, if not otherwise stated, was ICC 17.0.1 with the flags `-O3`, `-xHost`, `-ipo`, `-ansi-alias`, and `-fp-model precise`.

The experiments were conducted using both OpenTuner and HyperOpt. However, the former was more likely to find the best variant faster due to a more efficient meta-technique implementation, and by avoiding re-assessing variants already evaluated.

A. Matrix-Matrix Multiplication

We present here results for the optimizations in Figure 7 applied to the baseline code shown in Figure 3. The baseline code is a naive implementation of the double-precision matrix-matrix multiplication (DGEMM) and also served as input for Pluto and Locus. Its execution time on a single core was used to calculate the speedups. The three matrices involved have a 2048 by 2048 shape. Figure 7 represents an optimization space of 34,012,224 possible variants (according to OpenTuner).

The code in Figure 7 applies loop interchange and a two-level hierarchical tiling. Different tile sizes were explored at each level using 6 search variables (`tileI`, `tileI_2`, `tileK`, `tileK_2`, `tileJ`, and `tileJ_2`) that vary across all powers of two between 2 and 512. The best result used a tiling shape on the upper level represented by $I = 512$, $K = 256$, and $J = 32$; and on the lower level by $I_2 = 8$, $K_2 = 8$, and $J_2 = 128$. We apply an OpenMP pragma on the outermost loop and use an OR block to explore the space of optimization that the `parallel for` pragma provides: scheduling and chunk. The best variant, however, used the default values of the `parallel for`.

The right side of Figure 6 presents results from 1 to 10 CPU cores. They are compared to code generated by Pluto (flags `-tile`, `-l2tile`, and `-parallel`) and to Intel MKL 2017.0.1. The Locus search was limited to 1,000 variants for each case

```
#pragma @Locus loop=heat2d
for (t = 0; t < T; t++)
  for (i = 1; i < N+1; i++)
    for (j = 1; j < N+1; j++)
      A[(t+1)%2][i][j] = 0.125 * (A[t%2][i+1][j]
      - 2.0 * A[t%2][i][j] + A[t%2][i-1][j])
      + 0.125 * (A[t%2][i][j+1]
      - 2.0 * A[t%2][i][j] + A[t%2][i][j-1])
      + A[t%2][i][j];
```

Fig. 8. Heat 2D stencil kernel.

```
Search {
  buildcmd = "make␣clean;␣make";
  runcmd = "./heat-2d";
}
CodeReg heat2d{
  skew1 = poweroftwo(16..128);
  tmat = [[ skew1,     0,     0],
          [-skew1, skew1,     0],
          [-skew1,     0, skew1]];
  Pips.GenericTiling(loop="0", factor=tmat);
  Pragma.Ivdep(loop="0.0.0.0.0.0");
  Pragma.Vector(loop="0.0.0.0.0.0");
}
```

Fig. 9. Locus program for optimizing Heat 2D stencil.

and took on average 80 minutes to complete. Pluto generated code in less than a second. The code generated by Locus using 10 cores was 553 times faster than the baseline. Intel MKL was faster than Locus when using 1, 2, and 6 cores, but slower for 4, 8, and 10 cores. On average, the best variant generated by Locus was 3.45 times faster than the code generated by Pluto. The reason for the performance difference is not the set of transformations applied, Pluto and Locus apply the same transformations, but the use of empirical search to identify the best tile sizes.

B. Stencil Kernels

We evaluated tiling transformations using Locus on 6 stencil codes: Jacobi 1D and 2D, Heat 1D and 2D, and Seidel 1D and 2D[1]. The stencils were executed for 1,000 time steps; the dimensions of the 2D versions are 2,000 by 2,000 elements, and of the 1D are 1,600,000 elements (double-precision). As an example, Figure 8 presents the Heat 2D baseline version, and Figure 9 the optimizations applied.

We applied a Skewing-1 [18] tiling shape using Pips' *GenericTiling*. In the same way as Pluto, pragmas for improving vectorization are inserted before the innermost loops.

Pluto generated code in less than a second. Locus execution time (including the empirical search) for the stencil Heat 2D was the longest at 9 minutes. Jacobi 1D and 2D, Heat 1D, Seidel 1D and 2D lasted 3, 6, 2, 2, and 6 minutes respectively.

The Locus-generated code outperforms that of Pluto (flags `-tile` and `-pet`) as shown in Figure 6. Once again the empirical search showed its importance on the process, as the set of transformations applied by both was the same.

[1]The stencil codes were based on the examples provided by Pluto (*pet* branch) and can be found at http://pluto-compiler.sourceforge.net.

```
#pragma @Locus loop=Scattering
for(int nm = 0; nm < num_moments; ++nm)
   for(int g = 0; g < num_groups; ++g)
      for(int gp = 0; gp < num_groups; ++gp)
         for(int zone = 0; zone < num_zones; ++zone)
            for(int mix = zones_mixed[zone];
          mix < zones_mixed[zone] + num_mixed[zone]; ++mix) {
                int material = mixed_material[mix];
                double fraction = mixed_fraction[mix];
                int n = moment_to_coeff[nm];

                #####
                # Address calculation to be included here.
                #####

                *phi_out += *sigs * *phi * fraction;
          }
```

Fig. 10. Kripke's Scattering kernel.

```
datalayout=enum("DZG","DGZ","GDZ","GZD","ZDG","ZGD");
CodeReg Scattering {
  if (datalayout == "DGZ") {
     omploop="0.0.0.0";
  } elif (datalayout == "GDZ") {
     looporder=[1,2,0,3,4];
     omploop="0.0.0.0";
  } elif (datalayout == "GZD") {
     looporder=[1,2,3,4,0];
     omploop="0.0.0";
  } elif (datalayout == "ZGD") {
     looporder=[3,4,1,2,0];
     omploop="0";
  } elif (datalayout == "ZDG") {
     looporder=[3,4,0,1,2];
     omploop="0";
  } elif (datalayout == "DZG") {
     looporder=[0,3,4,1,2];
     omploop="0.0";
  }
  sourcepath="scatter_"+datalayout+".txt";
  BuiltIn.Altdesc(stmt="0.0.0.0.0.3", source=sourcepath);
  RoseLocus.Interchange(order=looporder);
  RoseLocus.LICM();
  RoseLocus.ScalarRepl();
  Pragma.OMPFor(loop=omploop);
}
```

Fig. 11. Locus program for optimizing Kripke's Scattering kernel.

C. Kripke

Kripke [19] is a deterministic particle transport code and a proxy-app for the Ardra project developed at LLNL. It supports storage of angular fluxes using a three dimensional array indexed by direction (D), group (G), and zone (Z). It has 5 kernels: *LTimes*, *LPlusTimes*, *Scattering*, *Source*, and *Sweep*. Their implementation contains 6 hand-optimized versions of each kernel, one for each data layout. Each layout corresponds to a different linearization of the 3D arrays according to one of the 6 permutations of D, G, and Z.

Using Locus, we can create a more compact representation of all versions of Krypke. Our representation contains only one skeleton for each kernel plus the code for each of the six address computations, one for each data layout. We then use transformation modules to generate versions that achieve a performance comparable to that of the hand-optimized ones.

Figure 10 contains the code representing our version of one of the five kernels, *Scattering*, and Figure 11 contains the Locus

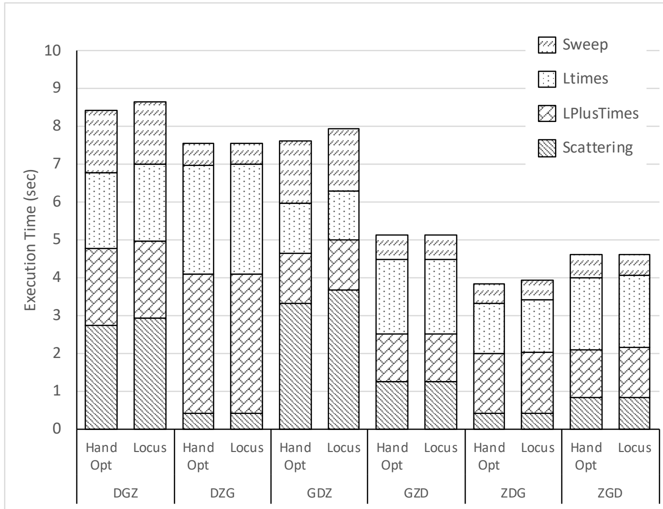

Fig. 12. Kripke execution time comparing hand optimized versions and using Locus for 6 different data layouts.

TABLE I
THE BENCHMARKS USED FOR LOOP EXTRACTION, THE NUMBER OF LOOP NESTS SELECTED AND VARIANTS ASSESSED.

Benchmark	# of loop nests	Variants assessed
ALPBench [23]	13	39
ASC Sequoia [24]	1	3
Cortexsuite [25]	47	1,297
FreeBench [26]	30	431
Parallel Research Kernels [27]	37	1,055
Livermore Loops [28]	11	121
MediaBench [29]	39	159
Netlib [30]	18	260
NAS Parallel Benchmarks [31]	208	23,384
Polybench [32]	93	7,582
Scimark2 [33]	4	83
SPEC2000 [34]	71	2,228
SPEC2006 [35]	50	216
Extended TSVC [36]	156	6,943
Libraries [37]–[40]	61	1,966
Neural Network Kernels [41]	17	132
Total	856	45,899

program to create the six versions of the kernel. These will differ in the evaluation of the addresses used by the expression (`*phi_out += *sigs * *phi * fraction`) at the end of the loop in Figure 10.

The main structure of the other four kernels (*LTimes*, *LPlusTimes*, *Source*, and *Sweep*) is very close to the one shown for *Scattering* and the Locus program for each of these kernels is similar to that in Figure 11.

The Locus program specifies that, for *Scattering*, the name of each of the six data layouts (*DZG*, *DGZ*, *GZD*, *ZDG*, and *ZGD*) will be used to select one of six different files (`scatter_DZG.txt`, `scatter_DGZ.txt`, ..., and `scatter_ZGD.txt`). Each of these files contains one way of computing the addresses depending on the layout. Different versions of *Scattering* are created by inserting the content of each of these files into the innermost loop of the kernel using the module `BuiltIn.Altdesc`. The specific place of insertion is after the third statement of the innermost loop which is identified as `0.0.0.0.0.3`. Then:

- the loop nest order (`looporder`) is altered by the module `RoseLocus.Interchange`;
- loop invariant code motion is applied by transformation module `RoseLocus.LICM` to move each part of the computation to the most efficient location within the loop nest;
- scalar replacement [20] is performed to improve register usage by the module `RoseLocus.ScalarRepl`;
- the loop to be parallelized (`omploop`) is annotated with OpenMP directives by the module `Pragma.OMPFor`.

The data layout was the only search variable defined. Other possibilities, however, could have been added to the search space. For instance, we used the loop nest orders identical to the hand-optimized versions, but it would be straightforward to explore different orders for each data layout.

Performance attained using Locus is very close to that of the hand-optimized kernels, as presented on Figure 12, but our proposed version is simpler and easier to maintain, because it contains a single version of each kernel for all data layouts.

D. Optimization of Arbitrary Loop Nests

We have shown performance improvements on optimizations specific to source codes known beforehand. On this experiment, we used a generic Locus program to optimize arbitrary loop nests, whose structure and characteristics were not known in advance. Gong et al. [21], [22] developed a loop nest extractor and transformed the extracted loops with subsets of the following two sequences:

1) interchange → unroll-and-jam → distribution → unrolling;
2) interchange → tiling → distribution → unrolling.

We represented these optimization sequences and their subsets in a Locus program, shown in Figure 13. The optimizations have a data dependence check and only legal transformations were used to generate variants. The first step is to check if the data dependences can be computed for the loop nest (if not, only unrolling is applied). The next step is to check whether the loop is perfectly nested. Then, the loop nest depth is obtained. These data are used to define which transformations are going to be applied next. Interchange is only applied to perfectly nested loops with nest depth greater than 1. Tiling is only applied on perfectly nested loops, and unroll-and-jam only on nests with depth greater than 1. Along with distribution, these are only applied if the data dependences can be computed. Unrolling is always applied at the end of the process.

An OR construct is used to express the choice among tiling, unroll-and-jam, or none before distribution. It is also important to note the $*$ on the distribution, which denotes that it is optional.

Table I presents the number of the loop nests extracted and the number of variants assessed for each benchmark. In total, 3,146 loop nests were extracted, and we selected 856 whose

```
Search {
  buildcmd = "make␣clean;␣make␣LOOPEXTRACTED";
  runcmd = "LOOPEXTRACTED␣../input␣10";
}
CodeReg scop {
  perfect = BuiltIn.IsPerfectLoopNest();
  depth = BuiltIn.LoopNestDepth();
  if (RoseLocus.IsDepAvailable()) {
    if (perfect && depth > 1) {
      permorder = permutation(seq(0,depth));
      RoseLocus.Interchange(order=permorder);
    }
    {
      if (perfect) {
        indexT1 = integer(1..depth);
        T1fac = poweroftwo(2..32);
        RoseLocus.Tiling(loop=indexT1, factor=T1fac);
      }
    } OR {
      if (depth > 1) {
        indexUAJ = integer(1..depth-1);
        UAJfac = poweroftwo(2..4);
        RoseLocus.UnrollAndJam(loop=indexUAJ,
                               factor=UAJfac);
      }
    } OR {
      None; # No tiling, interchange, or unroll and jam.
    }
    innerloops = BuiltIn.ListInnerLoops();
    *RoseLocus.Distribute(loop=innerloops);
  }
  innerloops = BuiltIn.ListInnerLoops();
  RoseLocus.Unroll(loop=innerloops,
                   factor=poweroftwo(2..8));
}
```

Fig. 13. Locus program for optimizing arbitrary loop nests.

execution are longer than 10,000 CPU cycles. The search for each loop nest was limited to 500 variants. In total 45,899 variants were evaluated. We run Pluto on the same set of selected loop nests.

Our work was able to reproduce the performance results presented by Gong et al. However, the Locus program required 37 lines while the implementation by Gong et al. for this purpose was approximately 1200 lines long. Besides, on their approach, the implementation of the two optimization sequences, their subsets, and parameters were hard-coded and cumbersome to modify. With the Locus approach, modifying and experimenting new optimization sequences becomes trivial.

In this experiment, the variants generated by Locus and the code generated by Pluto (flags `-tile`, `-prevector`, and `-unroll`) were compiled with GCC 6.3.0 (flags `-O3`, and `-ftree-vectorize`).

On average, the best variant generated by Locus achieved a speedup of 1.15 while Pluto achieved 1.05. Locus could transform 822 loop nests out of the ones selected and Pluto 397. Pluto transformed a smaller number of loop nests because it is based on the polyhedral model. This model is only applicable to loop nests in which the data access functions and the loop bounds are affine combinations of the enclosing loop variables and parameters. Locus achieved speedups higher than 1.05 for 360 loops nests and Pluto for 170. Out of these 170 optimized by both tools, Locus generated faster code than Pluto on 129 of them.

VI. Related Work

Several frameworks expose a high-level interface for code transformations. URUK [42] features loop transformations with unimodular schedules through a script that operates on a compiler intermediate representation. Loopy [12] carries out transformations defined by the composition of operations from a script on tagged loops. It includes a verifier that guard programmers against incorrect specifications.

CHiLL [43] contains loop transformations 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 [44] is an embedded scripting language for parameterizing AST-based transformations so that they can be empirically tuned. Clay [45] provides a transformation set to allow representing arbitrary polyhedral optimizations in a separate script or embedded as comments. The Locus language, however, is able to better represent complex optimization spaces on applications that have multiple code regions to be optimized.

Orio [4], 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 [46] provides pragmas that can perform loop transformations and code transformations defined as pattern-replacement rules. Locus implements a separation of concerns and avoids the definition of the optimizations along with the application code. Besides, our approach also abstracts the empirical search process, making it easier to compare different methods.

TunedCnC [47] is focused on separation of concerns through a declarative tuning framework for improving spatial and temporal locality on shared-memory and distributed systems. The many possible groupings of computational steps could be represented using Locus to search for optimal process placement.

Lift [10], [48] exploits functional principles to generate high-performance GPU code. Applications are expressed using a small set of functional primitives and optimizations are all encoded as formal, semantics-preserved rewrite rules. These rules define an optimization space that is automatically searched for high-performance code. Locus could be adapted to represent this optimization space and carry out the search.

Other systems focused on the optimization of a specific set of algorithms. FFTW [49] 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. ATLAS [11] 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 fits in the fastest level cache.

The goal of PHiPAC [50] is to produce high-performance linear algebra libraries for a wide range of systems with mini-

mum effort. The authors developed parameterized generators that produce code according to guidelines from a generic model of a set of C compilers and microprocessors.

SPIRAL [9] is another autotuning system used for digital signal processing. It has 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.

OSKI [51] is a collection of low-level primitives that provide automatically tuned computational kernels on sparse matrices. It defers the tuning until the run time to make decisions about the data structures and code transformations to be used.

VII. Conclusions and Future Work

In this paper, we presented Locus, a new system and a language for optimizing complex, long-lived applications for different environments. As architecture-specific optimizations are required to harness performance available on current machines, the Locus system is able to integrate different tools and combine expert knowledge with automated transformations to assist performance experts and code developers in the performance optimization process. We presented 4 examples to illustrate the simplicity of the language and its power to represent spaces of program variants resulting from complex transformation sequences.

These examples also show that using empirical search, it is possible to obtain higher speedups over a baseline version compared to conventional restructurers. The results, which include 553x on matrix-matrix multiplication and of up to 4x on stencil computations, match the hand-optimized performance for the Kripke transport code, and show good performance improvements for a collection of loop nests extracted from 16 benchmarks.

As future work, we plan to combine the use of multiple search modules in the same run to speed up the search process. Ongoing work aims to help users at designing optimization sequences.

Acknowledgment

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 1533912. We are grateful to the anonymous reviewers who helped improve the quality of the the paper. We also gratefully acknowledge Gong Zhangxiaowen and Justin Szaday for their valuable help in setting up the experiments presented in Section V-D.

References

[1] A. Venkat, M. Hall, and M. Strout, "Loop and data transformations for sparse matrix code," in Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, ser. PLDI '15. New York, NY, USA: ACM, 2015, pp. 521–532. [Online]. Available: http://doi.acm.org/10.1145/2737924.2738003

[2] M. S. Lam and M. E. Wolf, "A data locality optimizing algorithm," SIGPLAN Not., vol. 39, no. 4, pp. 442–459, Apr. 2004. [Online]. Available: http://doi.acm.org/10.1145/989393.989437

[3] N. Manjikian and T. S. Abdelrahman, "Fusion of loops for parallelism and locality," IEEE Transactions on Parallel and Distributed Systems, vol. 8, no. 2, pp. 193–209, Feb 1997.

[4] A. Hartono, B. Norris, and P. Sadayappan, "Annotation-based empirical performance tuning using orio," in IPDPS'09, 2009.

[5] K. Fatahalian, T. J. Knight, M. Houston, M. Erez, D. R. Horn, L. Leem, J. Y. Park, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan, "Sequoia: Programming the memory hierarchy," in SC 2006 Conference, Proceedings of the ACM/IEEE, Nov 2006, pp. 4–4.

[6] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe, "Petabricks: A language and compiler for algorithmic choice," in ACM SIGPLAN Conference on Programming Language Design and Implementation, Dublin, Ireland, Jun 2009. [Online]. Available: http://groups.csail.mit.edu/commit/papers/2009/ansel-pldi09.pdf

[7] J. Lidman, D. J. Quinlan, C. Liao, and S. A. McKee, "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), June 2012, pp. 1–6.

[8] R. Keryell, C. Ancourt, F. Coelho, F. Irigoin, and P. Jouvelot, "Pips: a workbench for building interprocedural parallelizers, compilers and optimizers," MINES ParisTech, Tech. Rep., 06 1996.

[9] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo, "SPIRAL: Code generation for DSP transforms," Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation", vol. 93, no. 2, pp. 232– 275, 2005.

[10] M. Steuwer, T. Remmelg, and C. Dubach, "Lift: A functional data-parallel ir for high-performance gpu code generation," in 2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), Feb 2017, pp. 74–85.

[11] R. C. Whaley and J. J. Dongarra, "Automatically tuned linear algebra software," in Proceedings of the 1998 ACM/IEEE Conference on Supercomputing, ser. SC '98. Washington, DC, USA: IEEE Computer Society, 1998, pp. 1–27. [Online]. Available: http://dl.acm.org/citation.cfm?id=509058.509096

[12] K. S. Namjoshi and N. Singhania, "Loopy: Programmable and formally verified loop transformations," in Static Analysis, X. Rival, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 383–402.

[13] J. C. Lévesque, A. Durand, C. Gagné, and R. Sabourin, "Bayesian optimization for conditional hyperparameter spaces," in 2017 International Joint Conference on Neural Networks (IJCNN), May 2017, pp. 286–293.

[14] L. Dagum and R. Menon, "Openmp: an industry standard api for shared-memory programming," IEEE Computational Science and Engineering, vol. 5, no. 1, pp. 46–55, Jan 1998.

[15] 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. [Online]. Available: http://doi.acm.org/10.1145/2628071.2628092

[16] J. Bergstra, D. Yamins, and D. D. Cox, "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, ser. ICML'13. JMLR.org, 2013, pp. I–115–I–123. [Online]. Available: http://dl.acm.org/citation.cfm?id=3042817.3042832

[17] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral parallelizer and locality optimizer," in Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation, ser. PLDI '08. New York, NY, USA: ACM, 2008, pp. 101–113. [Online]. Available: http://doi.acm.org/10.1145/1375581.1375595

[18] X. Zhou, M. J. Garzarán, and D. A. Padua, "Optimal parallelogram selection for hierarchical tiling," ACM Trans. Archit. Code Optim., vol. 11, no. 4, pp. 58:1–58:23, Jan. 2015. [Online]. Available: http://doi.acm.org/10.1145/2687414

[19] A. J. Kunem, P. Brown, T. S. Bailey, and P. G. Maginot, "Kripke: 3d sn deterministic particle transport code," https://computation.llnl.gov/projects/co-design/kripke, 2014.

[20] K. Kennedy and J. R. Allen, Optimizing Compilers for Modern Architectures: A Dependence-based Approach. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002.

[21] Z. Gong, Z. Chen, J. Szaday, D. Wond, Z. Sura, N. Watkinson, S. Maleki, D. Padua, A. Veidenbaum, A. Nicolau, and J. Torrellas, "An empirical study of the effect of source-level loop transformations on compiler stability," Proc. ACM Program. Lang., vol. 1, no. OOPSLA, Oct. 2018.

[22] Z. Chen, Z. Gong, J. J. Szaday, D. C. Wong, D. Padua, A. Nicolau, A. V. Veidenbaum, N. Watkinson, Z. Sura, S. Maleki, J. Torrellas, and G. DeJong, "Lore: A loop repository for the evaluation of compilers," in 2017 IEEE International Symposium on Workload Characterization (IISWC), Oct 2017, pp. 219–228.

[23] M.-L. Li, R. Sasanka, S. V. Adve, Y.-K. Chen, and E. Debes, "The alpbench benchmark suite for complex multimedia applications," in IEEE International. 2005 Proceedings of the IEEE Workload Characterization Symposium, 2005., Oct 2005, pp. 34–45.

[24] LLNL, "Asc sequoia benchmark," https://asc.llnl.gov/sequoia/benchmarks/, 2008.

[25] S. Thomas, C. Gohkale, E. Tanuwidjaja, T. Chong, D. Lau, S. Garcia, and M. B. Taylor, "Cortexsuite: A synthetic brain benchmark suite," in 2014 IEEE International Symposium on Workload Characterization (IISWC), Oct 2014, pp. 76–79.

[26] P. Rundberg and F. Warg, "The freebench v1.0 benchmark suite," http://www.freebench.org, 2002.

[27] R. F. V. der Wijngaart and T. G. Mattson, "The parallel research kernels," in 2014 IEEE High Performance Extreme Computing Conference (HPEC), Sept 2014, pp. 1–6.

[28] T. Peters, "Livermore loops coded in c," http://www.netlib.org/benchmark/livermorec, 1992.

[29] J. E. Fritts, F. W. Steiling, J. A. Tucek, and W. Wolf, "Mediabench ii video: Expediting the next generation of video systems research," Microprocessors and Microsystems, vol. 33, no. 4, pp. 301 – 318, 2009, media and Stream Processing. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S014193310900026X

[30] J. Dongarra, G. H. Golub, E. Grosse, C. Moler, and K. Moore, "Netlib and na-net: Building a scientific computing community," IEEE Annals of the History of Computing, vol. 30, no. 2, pp. 30–41, April 2008.

[31] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga, "The nas parallel benchmarks—summary and preliminary results," in Proceedings of the 1991 ACM/IEEE Conference on Supercomputing, ser. Supercomputing '91. New York, NY, USA: ACM, 1991, pp. 158–165. [Online]. Available: http://doi.acm.org/10.1145/125826.125925

[32] L.-N. Pouchet, "Polybench: The polyhedral benchmark suite." http://web.cse.ohio-state.edu/ pouchet.2/software/polybench/, 2012.

[33] B. M. Roldan Pozo, "Scimark 2.0," http://math.nist.gov/scimark2, 2004.

[34] J. L. Henning, "Spec cpu2000: measuring cpu performance in the new millennium," Computer, vol. 33, no. 7, pp. 28–35, July 2000.

[35] ——, "Spec cpu2006 benchmark descriptions," SIGARCH Comput. Archit. News, vol. 34, no. 4, pp. 1–17, Sep. 2006. [Online]. Available: http://doi.acm.org/10.1145/1186736.1186737

[36] S. Maleki, Y. Gao, M. J. Garzarán, T. Wong, and D. A. Padua, "An evaluation of vectorizing compilers," in Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques, ser. PACT '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 372–382. [Online]. Available: http://dx.doi.org/10.1109/PACT.2011.68

[44] Q. Yi, K. Seymour, H. You, R. Vuduc, and D. Quinlan, "Poet: Parameterized optimizations for empirical tuning," in 2007 IEEE International Parallel and Distributed Processing Symposium, March 2007, pp. 1–8.

[37] GAP, "Groups, algorithms, programming - a system for computational discrete algebra." www.gap-system.org, 2007.

[38] LAME, "Lame mp3 encoder." lame.sourceforge.net, 2017.

[39] Mozilla, "Mozilla jpeg encoder project." github.com/mozilla/mozjpeg, 2017.

[40] TwoLAME, "Twolame - mpeg audio layer 2 encoder," github.com/mozilla/mozjpeg, 2017.

[41] J. Redmon, "Darknet: Open source neural networks in c," http://pjreddie.com/darknet/, 2013–2016.

[42] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, and O. Temam, "Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies," International Journal of Parallel Programming, vol. 34, no. 3, pp. 261–317, Jun 2006. [Online]. Available: https://doi.org/10.1007/s10766-006-0012-3

[43] C. Chen, J. Chame, and M. Hall, "Chill: A framework for composing high-level loop transformations," University of Utah, Tech. Rep., 2008.

[45] L. Bagnères, O. Zinenko, S. Huot, and C. Bastoul, "Opening polyhedral compiler's black box," in Proceedings of the 2016 International Symposium on Code Generation and Optimization, ser. CGO '16. New York, NY, USA: ACM, 2016, pp. 128–138. [Online]. Available: http://doi.acm.org/10.1145/2854038.2854048

[46] S. Donadio, J. Brodman, T. Roeder, K. Yotov, D. Barthou, A. Cohen, M. J. Garzarán, D. Padua, and K. Pingali, "A language for the compact representation of multiple program versions," in Languages and Compilers for Parallel Computing, E. Ayguadé, G. Baumgartner, J. Ramanujam, and P. Sadayappan, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 136–151.

[47] S. Chatterjee, N. Vrvilo, Z. Budimlic, K. Knobe, and V. Sarkar, "Declarative tuning for locality in parallel programs," in 2016 45th International Conference on Parallel Processing (ICPP), Aug 2016, pp. 452–457.

[48] M. Steuwer, C. Fensch, S. Lindley, and C. Dubach, "Generating performance portable code using rewrite rules: From high-level functional expressions to high-performance opencl code," in Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ser. ICFP 2015. New York, NY, USA: ACM, 2015, pp. 205–217. [Online]. Available: http://doi.acm.org/10.1145/2784731.2784754

[49] M. Frigo, "A fast fourier transform compiler," in Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation, ser. PLDI '99. New York, NY, USA: ACM, 1999, pp. 169–180. [Online]. Available: http://doi.acm.org/10.1145/301618.301661

[50] J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel, "Optimizing matrix multiply using phipac: A portable, high-performance, ansi c coding methodology," in Proceedings of the 11th International Conference on Supercomputing, ser. ICS '97. New York, NY, USA: ACM, 1997, pp. 340–347. [Online]. Available: http://doi.acm.org/10.1145/263580.263662

[51] R. Vuduc, J. W. Demmel, and K. A. Yelick, "Oski: A library of automatically tuned sparse matrix kernels," Journal of Physics: Conference Series, vol. 16, no. 1, p. 521, 2005. [Online]. Available: http://stacks.iop.org/1742-6596/16/i=1/a=071