The Sparse Polyhedral Framework: Composing Compiler-Generated Inspector–Executor Code

By Michelle Mills Strout, Member IEEE, Mary Hall, Senior Member IEEE, and Catherine Olschanowsky, Member IEEE

ABSTRACT | Irregular applications such as big graph analysis, material simulations, molecular dynamics simulations, and finite element analysis have performance problems due to their use of sparse data structures. Inspector–executor strategies improve sparse computation performance through parallelization and data locality optimizations. An inspector reschedules and reorders data at runtime, and an executor is a transformed version of the original computation that uses the newly reorganized schedules and data structures. Inspector–executor transformations are commonly written in a domain-specific or even application-specific fashion. Significant progress has been made in incorporating such inspector–executor transformations into existing compiler transformation frameworks, thus enabling their use with compile-time transformations. However, composing inspector–executor transformations in a general way has only been done in the context of the Sparse Polyhedral Framework (SPF). Though SPF enables the general composition of such transformations, the resulting inspector and executor performance suffers due to missed specialization opportunities. This paper reviews the history and current state of the art for inspector–executor strategies and reviews how the SPF enables the composition of inspector–executor transformations. Further, it describes a research vision to combine this generality in SPF with specialization to achieve composable and high performance inspectors and executors, producing a powerful compiler framework for sparse matrix computations.

KEYWORDS | Intermediate representations; irregular computations; program optimization and parallelization; sparse matrices

I. INTRODUCTION

Irregular applications have moved to the forefront of scientific computing and data analytics. Molecular dynamics simulations, materials simulations, finite element analysis, and big graph analysis are some of the many important application domains that rely on efficient computation over sparse matrices or graphs and unstructured meshes. Sparse and unstructured computations reduce data storage and computation requirements by using indirect accesses through index arrays to store only nonzero data elements or use a fine grid only in areas of interest. For example, Fig. 1 shows a sparse matrix vector multiplication (SpMV) computation that has been written to operate on a compressed sparse row (CSR) matrix representation. In this example, nonzeros in the sparse matrix are stored by row with the index array rowptr indicating where the nonzeros for a particular row start and the index array col indicating the column of each nonzero. Because of indirect memory access patterns, these irregular computations are notoriously memory bound, and this problem is getting worse on modern architectures where cost of data movement dominates computation execution time. To address these performance challenges, in recent years
and the nonzeros are organized by row. The index arrays of the sparse matrix data structure is called compressed sparse row. The majority of the zeroes in the matrix are not stored. In this example, the inspector. In the SpMV computation in Fig. 1, the inspector can achieve high performance. Venkat et al. produced runtime mappings of computation overlap to improve data locality. To improve data locality in more than just individual loops, Strout et al. composed inspectors for data and iteration reordering with full sparse tiling, which groups iterations across loops. Wu et al. produce runtime mappings of computations to SMs for improved GPU performance of irregular applications.

Prior work has shown that automatically generated inspectors can achieve high performance. Venkat et al. [27], [28] use a combination of sparse data structure transformations and parallelization approaches to provide better performance than the hand-tuned CUSP and OSKI libraries for sparse matrix vector multiplication on a GPU and CPU, respectively. A more complex inspector for parallel conjugate gradient combines automatically generated runtime dependence testing with a programmer-defined breadth-first search library for deriving level sets of iterations that may be safely executed in parallel [29]. This inspector performs comparably to the hand-tuned

there has been a significant body of research on new sparse matrix libraries and data representations and their implementations that specialize for particular application domains and new architectures [11]–[8].

These algorithms and representations are commonly encapsulated in manually tuned, architecture-specific libraries such as Intel’s MKL or Nvidia’s cuSPARSE. A purely library approach has several weaknesses: 1) there are dozens of sparse matrix/graph algorithms and representations in common use, many new ones being developed, and the code for each must be manually optimized; 2) libraries must be manually ported to new architectures; and 3) a library encapsulates individual functions that cannot be composed in an application (e.g., only performs a single sparse matrix–vector multiply or computation of similar scope). Ideally, programmer productivity and code efficiency would be greatly enhanced by appropriate abstractions and tools to simplify the development of sparse matrix applications.

In many of these codes, the sparse structure of the matrix or graph does not change through all or significant intervals of the computation. Thus inspector–executor strategies were developed to parallelize and improve the data locality of such computations [9]. At runtime, the inspector code traverses the index arrays to determine data access patterns and their resulting data dependencies. This runtime information can then be used to derive schedules and reordered data structures that the executor uses to execute the computation in a more efficient manner. Schedules in this context can be modified by introducing another layer of indirection and/or modifying an existing level of indirection with the new index array or reordering of an existing index array being performed at runtime by the inspector. In the SpMV computation in Fig. 1, the current schedule is to iterate over the matrix in row order.

If instead the order to compute rows was specified by a permutation array (i.e., row = p[i] instead of row = i), then the inspector could change the loop schedule at runtime by setting the values in the permutation array. A data reordering can be effected if: 1) the compiler generates code to access each reordered data array through a new index array; and 2) the runtime inspector reorders the data arrays and populates the new index arrays appropriately. For example, y[i] would become y’[b[i]] at compile time and the inspector would reorder the y array into y’ and populate the index array b[] to reflect that reordering. This additional indirection resulting from optimization even further complicates applying additional optimizations.

This paper describes the state-of-the-art and future research on compiler technology to achieve this goal. Compilation techniques have been developed to create a distributed memory parallelization of codes with indirect array accesses [9]–[12], find wavefronts of parallelism in codes such as sparse triangular solves [13]–[16], to automate the conversion of dense matrix code to specialized sparse matrix code [17]–[19], effect the reordering of data and iterations in reduction or parallel loops [20]–[24], the reordering of iterations to improve communication and computation overlap [25], [26], and combining inspector–executor transformations with polyhedral transformations [26]–[29]. Some of these approaches illustrate that automation of particular inspector–executor strategies in combination with existing loop transformations is possible.

Specific inspector combinations have been found to be important for the performance of the resulting executor and therefore have been automated. Ding and Kennedy composed data array permutation and iteration permutation inspectors [20] to improve spatial and temporal data locality. Lee and Eigenmann [25] and Ravishankar et al. [26] combined iteration reordering within each loop with computation and communication overlap to improve data locality. To improve data locality in more than just individual loops, Strout et al. [30], [31] composed inspectors for data and iteration reordering with full sparse tiling, which groups iterations across loops. Wu et al. [32] produce runtime mappings of computations to SMs for improved GPU performance of irregular applications.

Prior work has shown that automatically generated inspectors can achieve high performance. Venkat et al. [27], [28] use a combination of sparse data structure transformations and parallelization approaches to provide better performance than the hand-tuned CUSP and OSKI libraries for sparse matrix vector multiplication on a GPU and CPU, respectively. A more complex inspector for parallel conjugate gradient combines automatically generated runtime dependence testing with a programmer-defined breadth-first search library for deriving level sets of iterations that may be safely executed in parallel [29]. This inspector performs comparably to the hand-tuned

Fig. 1. The sparse matrix vector multiplication computation operates on a sparse matrix data structure where either all or the majority of the zeroes in the matrix are not stored. In this example, the sparse matrix data structure is called compressed sparse row and the nonzeros are organized by row. The index arrays rowptr and col provide information about where in memory the nonzeros for a particular row start and for each nonzero what its column is, respectively.

implified breadth-first search library for deriving level sets of iterations that may be safely executed in parallel [29]. This inspector performs comparably to the hand-tuned

Prior work has shown that automatically generated inspectors can achieve high performance. Venkat et al. [27], [28] use a combination of sparse data structure transformations and parallelization approaches to provide better performance than the hand-tuned CUSP and OSKI libraries for sparse matrix vector multiplication on a GPU and CPU, respectively. A more complex inspector for parallel conjugate gradient combines automatically generated runtime dependence testing with a programmer-defined breadth-first search library for deriving level sets of iterations that may be safely executed in parallel [29]. This inspector performs comparably to the hand-tuned
This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.


Fig. 2. The CHILL-I/E compiler will generate the executor, which is a transformed version of the original irregular computation. The compiler will also generate some subset of inspectors needed for nonaffine transformations and compose compiler-generated and programmer-defined inspectors into a single inspector.

SpMP library [15], as does the automatically generated executor.

The previously described optimization techniques were applied somewhat in isolation and for specific sequences of inspector–executor transformations. For example, although the Venkat et al. work automatically generated composed inspectors, the optimizations for the composed inspector were hardcoded for the specific sequence of transformations involved. What has been missing is a more general compiler framework and associated abstractions capable of composing a collection of inspector–executor transformations.

In this paper, we describe the Sparse Polyhedral Framework (SPF), which enables the composition of inspector–executor transformations with each other and with other loop transformations [30], [31]. As is implied by the name, SPF is an extension to the polyhedral framework and provides a way to represent sparse codes and transformations on sparse codes; loop transformations of both the inspector–executor and polyhedral variety. SPF represents loop iteration spaces as linear constraints plus constraints involving uninterpreted functions that represent nonaffine values; transformations are described as relations on these iteration spaces, and code generation employs polyhedra scanning plus some manipulation of uninterpreted functions of the transformed iteration spaces after composing sequences of transformations. Another key aspect of SPF is specifying how inspectors produce information at runtime (e.g., the number of iterations in a loop, a heuristic permutation for a data array, etc.) so that multiple inspectors can be composed.

In [31], a Python interface to SPF was used to specify sparse computations and various inspector–executor transformations of those computations. More recently, we have been incorporating SPF into the CHiLL compilation framework, which permits scripts called transformation recipes to describe the sequence of transformations to be composed (see Fig. 2). The system combines the SPF’s abstractions for supporting inspector–executor transformations, and CHiLL’s support for polyhedral transformation and code generation in the presence of sparse loop bounds and subscript expressions. This research is developing inspector–executor transformation specifications for sequences of transformations that were previously hard-coded in CHiLL [33].

The problem remains that compilation strategies that combine inspector–executor transformations with other loop-level transformations are still dealing with a tradeoff between the generality of the transformations that can be composed and the performance of the resulting inspectors and executors. To enable the general composition of inspectors, it is necessary to have more general index array implementations shared between inspectors and the final composed executor. The SPF does show promise in that the information needed to optimize composed inspectors is available to the compiler. In this paper, we review the prior art in providing compiler support for inspector–executor strategies, show how SPF enables the composition of inspectors and possible optimizations of those inspectors, and overview current research to incorporate this ability to create efficient composed inspectors into the CHiLL compilation tool.

II. BACKGROUND: COMPILER SUPPORT FOR INSPECTOR-EXECUTOR TRANSFORMATIONS

Inspector–executor transformations are most typically loop and data layout transformations that require a runtime component due to indirect array accesses that cannot be resolved until runtime. Compiler support for these transformations takes the form of the compiler inserting
calls to hand-written inspectors, generating inspectors for specific strategies, and transforming key loop nests into executors. In this section, we review the categories of inspector–executor transformations in roughly the order they were historically developed, which tracked introduction of architectural features that sparked the interest of the compiler community, so the section is ordered by category of optimizations. For each category we indicate some of the compilation technology that has been developed to effect efficient inspectors and executors. We highlight examples where inspector–executor transformations have been composed and end the section with compiler optimizations that reduce the overhead of inspection and improve the performance of the executor.

A. Challenges Posed by Sparse Codes

Sparse matrix and unstructured mesh computations pose many challenges to compilers and application developers alike. These challenges are illustrated by the simple SpMV code of Fig. 1. First, if we want to understand which locations of array \( x \) are being accessed, this information is unavailable at compile time. In the case of this simple code, we would want this information to identify data reuse of \( x \). In the general case where variables such as \( x \) are also written in the loop, we would want this information to determine the dependences that constrain parallelization of the computation. For either locality or parallelization, we want to collect data access information, and this may require information only available at runtime. This data access information could then be exploited to reorganize the data layout and/or computation to improve locality or manage parallelism. In addition, the inner loop of the computation has an unknown number of iterations, which can only be determined at runtime; thus, if we parallelize the computation at the outer loop level, then the work per thread may be imbalanced. This inner loop poses an additional challenge for parallelizing compilers, which mostly operate on computations in the affine domain, where subscript expressions and loop bounds are linear functions of surrounding loops.

To summarize, the challenge from a compiler perspective with nonaffine sparse matrix computations is the need for runtime information to be used in making optimization decisions. Runtime information is used for essentially two purposes:

- determining memory access patterns to determine data dependences for parallelization and locality optimizations;
- examining nonzero structure of sparse matrices to identify best data representation (i.e., way to organize the nonzeros).

For these purposes, the inspector–executor approach was introduced by Saltz and his collaborators [9], [34], [35]. An inspector collects runtime information regarding data and computation and, using this information, may also derive new data representations or schedules for the computation. An executor then applies the information gathered and optimizations planned by the inspector.

B. Distributed Memory Parallelization

The earliest work in this area focused on applying an inspector to collect dependence information at runtime that was used by the executor to guide distributed memory parallelization [9]. Soon afterward, inspector–executor was applied to the problem of what was called runtime compilation for distributed memory architectures [35]–[38]. Inspectors determined the processors where each data element resides using the owner computes rule, and collected communication schedules, loop iteration partitions, and other information that associates off-processor data copies with on-processor buffer locations. Executors could then gather nonlocal data, perform the computation, and then, if necessary, scatter the results. In the CHAOS compiler, an optimization applied to inspectors used dataflow analysis to detect when inspectors can be reused within a computation [35].

C. Shared Memory Parallelization

Concurrently with this work on compilation for distributed memory architectures, improvements to combined static and runtime dependence testing for shared-memory multiprocessors was further evolving. Static dependence analysis must conservatively assume a dependence whenever it cannot prove otherwise. Rauchwerger and Padua developed the LRPD test, which speculatively identified opportunities for parallelization, reduction, and privatization using a compressed bit map that marked dependences at runtime [39]. If dependences were identified, then the speculative parallel computation would be rerun sequentially.

An alternative approach relied on compile-time analysis to develop quick runtime checks to determine before entering a loop whether a parallel variant could be executed. This worked by simplifying equations arising from dependence analysis and using the inverse of the simplified result to guard execution of a parallel computation [40], [41]. For example, if a dependence exists only if \( n > 100 \), then the parallel code could be guarded by the condition \( \text{if } (n \leq 100) \). Such approaches can be thought of as using very specialized and efficient inspectors.

Another topic of recent research interest is parallelization in the presence of dependences, or partial parallelism, where, for example, a subset of independent iterations of a loop can be executed between synchronization points. In [13], Rauchwerger surveyed various techniques for dynamically scheduling iterations into wavefronts such that all of the iterations within one wavefront may be executed in parallel. An inspector for detecting partial parallelism inspects all the dependences for a loop and places iterations into irregular wavefronts. There is also more recent
work using the partial parallelism strategy in manually optimized code [14], [15]. Our own most recent work combines dependence analysis simplification with wavefront parallelization of a parallel conjugate gradient code [29].

D. Data Locality Optimization

As memory systems started to become a performance bottleneck, researchers turned their attention to locality optimization for cache, even for sequential computation. Identifying data reuse and reorganizing data and computation for the memory hierarchy relied on an inspector–executor approach.

Various data and iteration reordering transformations improve the spatial and temporal data locality of sparse codes. Ding and Kennedy developed locality grouping, which reorganizes computation to improve reuse, while dynamic data packing reorganizes data to increase spatial locality [20]. Mitchell et al. derived bucket tiling, which combines loop permutation according to some access function of an array with indirect accesses, data reordering and loop regeneration [21]. GPART clustered data based on reuse, abstracting dependences as edges in a graph based on graph partitioning; the computation is then reordered in Z-curve order based on the graph representation [23].

The reordering transformations operated on individual loops. The sparse tiling transformations, unstructured cache blocking [42], full sparse tiling [43], [44], and communication avoiding rescheduling [45] improve temporal data locality across loops of sparse computations. Additionally, the sparse tiling inspector–executor transformations can be used to create coarse-grain parallelism by grouping across iterations in an outer loop or between loops [46]. Thus sparse tiling composed with wavefront parallelism is an example of when it is beneficial to compose inspector–executor transformations.

Much of the subsequent work on reorganizing data to improve locality focused on sparse matrix computation, and often identified a specialized sparse matrix format and associated computation structure that achieves much higher performance than a standard representation. For example, the commonly used compressed sparse row (CSR) representation as in Fig. 1 includes only the nonzero entries of the matrix, with two auxiliary arrays to reconstruct the corresponding row and column of each nonzero entry. The use of an inspector mimics sparse matrix and graph libraries, which often convert to these specialized matrix formats early in a computation to exploit whatever structure is present in the nonzeros of the input matrix [5], [8], [47]–[50]. As a specific example, the Block Compressed Sparse Row (BCSR) representation, widely used in libraries, inserts a small number of zero-valued elements into the sparse matrix representation, which has the effect of increasing computation but making memory access patterns and generated code more regular and efficient. Compiler approaches also performed this matrix format transformation in an inspector [22], [24], [28], [51]. These transformations introduce dense inner loops in the executor that can be optimized to great effect by general purpose compilers (e.g., scalar replacement, register allocation, and vectorization). While a format conversion is costly, the assumption is that the matrix structure is not changing in the application, and the inspector will execute once; therefore, the cost of the conversion is amortized over many iterations of the solver.

E. Data Locality and Parallelization

Over the last several years, there has been intense interest in combining some of the previously described inspector–executor optimizations with parallelization, to target the various and vast amounts of parallelism offered by modern architectures. Basumallik et al. developed sparse matrix optimizations in the context of a compiler that converts shared-memory parallel code to MPI implementations [11]. Ravishankar et al. optimized their inspector and executor code when affine loops are found in producer/consumer relationships with loops that have nonaffine array accesses [52]. In later work, Ravishankar et al. [26] composed their distributed memory parallelization inspector–executor transformation with affine transformations that enable vector optimization. Inspector–executor strategies like full sparse tiling, which breaks code into tiles over reused data, enable better scaling on the node due to reduced memory bandwidth demands [46], [53], [54]. Several recent approaches have employed custom matrix representations and parallelization strategies that target graphics processors [24], [27], [28].

F. Alternative Compiler Approaches

Some previous work developed compiler optimizations for sparse matrices beginning with a dense abstraction of a sparse matrix computation. These compilers generated sparse data representations during code generation [17]–[19], [55]. These compilers either incorporated a small, fixed set of matrix representations for which code generation is straightforward or relied on the programmer or user to provide implementations for accessing data in sparse formats for operations such as searching, enumeration and dereferencing. Shpeisman and Pugh [18] specified an intermediate program representation for transforming sparse matrix codes, which directs an underlying C++ library for efficient enumeration, permutation, and scatter-gather access of nonzeros stored according to some compressed stripe storage. The Bernoulli compiler permitted extension to new formats by abstracting sparse matrix computations into relational expressions that describe constraints on the iteration space and predicates to identify nonzero elements [17], [56], [57]. Using an approach similar to optimizing relational
database queries, the compiler derived a plan for efficiently evaluating the relational expressions, and generates corresponding code. Gilad et al. [19] used the LL functional language for expressing and verifying sparse matrix codes with their dense analogs, under the assumption that the matrix is dense initially.

Newer code generation approaches for sparse codes have been developed by Kjolstad et al. [58] and provided in the Tensor Algebra Compiler (taco). Sparse tensors are the multidimensional generalization of sparse matrices. The taco code generator takes a tensor expression such as matrix addition or tensor multiplication, the format each input tensor, and generates efficient code that uses those formats. This approach works well for computations that can be expressed as tensor expressions. Other code generation work provides compile-time inspection to perform code optimizations like unrolling and vectorization specific to a particular sparsity pattern [59]. This approach is generally applicable, but is currently implemented for specific algorithms such as Cholesky.

G. Inspector Code Generation and Optimization

To set the stage for the remainder of the paper, we discuss three specific aspects of compiler support for inspectors: 1) optimizations applied to inspectors; 2) broadening the applicability of inspector-executor transformations; and 3) code generation of inspectors.

From the beginning it was clear that inspector code would need to be amortized over the improvements experienced by preferably multiple executions of the executor code. Initially a sequential inspector used to parallelize the executor was usually the dominant cost in an irregular code, so soon parallelizing the inspectors became an important area of study [34], [60], [61]. Other research focused on where the inspector code could be inserted to minimize the number of times an inspector needs to execute [20], [38]. Han et al. [23] determined when new schedules and data reorderings did not need to change for correctness and could be allowed to degrade some in terms of performance. Partial redundancy elimination approaches were developed to only compute the portions of the new schedules necessary when index arrays were recomputed in applications like molecular dynamics simulations [38], [62]. Agrawal et al. [62] developed interprocedural analyses for finding the effective points to call inspectors. All of the above techniques are still relevant and should be incorporated into a system that generates efficient inspector and executor code.

Not only should the inspector be efficient, it is also important that inspector-executor transformations are broadly applicable. Das et al. [63] developed compilation techniques to flatten multiple levels of indirection into separate loops with one level of indirection to enable the more general application of distributed memory parallelization in the Fortran D compiler. Van der Spek [64] developed an approach for converting pointer-based data structure code into indirect array accesses thus making inspector-executor transformations available for such codes.

We also consider prior work where inspectors are generated from the original code. In our previous work, we have developed three code transformations—make-dense, compact, and compact-and-pad—which facilitate composing inspector generation and executor optimization with standard loop transformations [28]. The make-dense transformation takes as input a set of nonaffine array index expressions and introduces a guard condition and as many dense loops as necessary to replace the nonaffine index expressions with affine accesses; this transformation enables further affine loop transformations such as tiling. Compact and compact-and-pad are inspector-executor transformations; an automatically generated inspector gathers the iterations of a dense loop that are actually executed and the optimized executor only visits those iterations. The executor represents the transformed code that uses the compacted loop, which can then be further optimized. Using compact-and-pad, the inspector also performs a data transformation, inserting explicit zeros when necessary to correspond with the optimized executor. The make-dense transformation is similar to sublimation presented by van der Spek et al. [65]. Sublimation requires analysis or pragmas to determine injectivity properties of the access functions so that the sublimation transformation can replace an existing loop with irregular bounds (like the inner loop in SpMV) with the dense loop. Additionally, a related idea of expanding a sparse vector into a dense vector is called access pattern expansion [66]. The compact transformation is related to guard encapsulation [67], which moves tests for elements into the loop bounds.

In summary, many inspector-executor strategies have been developed with some of them being incorporated into research compilers. However, the incorporation of inspector-executor transformations into compilers has been specific to the transformations that were to be applied. This makes it difficult to compose inspector-executor transformations in any general way.

III. COMPOSING INSPECTOR-EXECUTOR TRANSFORMATIONS

Composing transformations is one of the strengths of loop transformation frameworks like the polyhedral framework. In the polyhedral framework, loop computations are represented with iteration spaces, mappings of iterations to data, and data dependences between iterations. Transformations are represented as mappings and can be composed using functional composition.

The SPF retains the strength of the polyhedral framework while representing information that is not available until runtime. Uninterpreted functions are used as

---

1http://tensor-compiler.org

2https://bitbucket.org/taknevski/pldi15.git
A. Loop Transformation Frameworks

Loop transformation frameworks enable composing transformations such as loop fusion, fission, skewing, permutation, and tiling. Such frameworks have enjoyed significant integration into general-purpose compilers such as GNU gcc with Graphite [68], IBM XL and LLVM with Polly [69]. There are also source-to-source tools such as Pluto [70] and PPCG [71] that implement such loop transformation frameworks and scheduling algorithms that automatically determine a sequence of loop transformations for a particular loop nest.

Transformation frameworks provide compile-time abstractions for representing computation and data and transformations on those computations and data [72]. Polyhedral transformation frameworks represent loop iteration spaces as integer tuple sets that are executed in lexicographical order (e.g., in Fig. 3 the original loop’s iteration space is \( I = \{0, j, 0|1 \leq j < N\} \cup \{1, k, 0|0 \leq i < N - 1\} \)) and represent data accesses as mappings from the iteration space to data spaces (e.g., \( A_{I \rightarrow B} = \{1, k, 0 \rightarrow [k + 1]\} \) for the B \([k + 1]\) read in Fig. 3). The data dependences between iterations that access the same data location are expressed as mappings between iterations.

In the polyhedral framework, transformations are expressed as functions that map integer tuples in an iteration or data space into a new iteration or data space

\[
\{i_0, i_1, \ldots, i_d\} \rightarrow \{i'_0, i'_1, \ldots, i'_e\} \ | \ \vec{i}' = f(\vec{i}),
\]

where the function \( f \) must be affine (i.e., linear plus a constant). The composition of transformations is done by function composition. Consecutive transformations need to have their input dimensionality match the output dimensionality of the previous transformation. A composed transformation on the iteration space is correct if it satisfies the partial ordering dictated by the data dependences.

Fig. 3 illustrates how a loop nest represented as a set of integer tuples can be transformed with loop fusion composed with loop shifting. One important thing to note is that loop bounds and memory accesses (e.g., B \([k + 1]\)) are transformed by using the transformation function to solve for the old iterators in terms of the new iterators.

In the fused loop in Fig. 3, the new iterator is \( j' \). Solving for \( j \) and \( k \) in terms of \( j' \) using the transformation function \( T \) results in \( j = j' \) and \( k = j' - 1 \). Substitution leads to the new loop bounds for the \( k \) loop that are identical to the original \( j \) loop bounds and to the updated array accesses B[\( j' \)], A[\( j' - 1 \)], and B[\( j' \)].

B. Uninterpreted Functions

The SPF extends the polyhedral framework with the use of uninterpreted functions to represent nonaffine array accesses, nonaffine loop bounds, and inspector–executor transformations. The idea of representing nonaffine array accesses with uninterpreted functions was originally introduced by Wonnacott and Pugh in the context of data dependence analysis [74]. SPF goes further by representing loop bounds, transformations, and more information about the uninterpreted functions.

An uninterpreted function is a function where the mapping of inputs to outputs is unknown. The advantage of using uninterpreted functions when representing code and transformations on code is that there is some information known about them. Every uninterpreted function satisfies functional consistency, which states that if the same inputs are given to the function then the same output will be produced

\[ x = y \Rightarrow f(x) = f(y). \]

In SPF, various information is provided about all uninterpreted functions provided. All index arrays that are inputs or outputs of inspectors need to have their domain and range specified. The domain describes the bounds on indices into the index array and the range indicates the possible values being stored in index arrays. Other information that is crucial for generating efficient code and in some cases any code at all is whether an
uninterpreted function will be bijective or monotonic at runtime. Our IEGenLib,\(^3\) which manipulates sets and relations with affine and uninterpreted function constraints, provides interfaces for specifying such information. Other research has shown that such information about index arrays can be helpful in finding parallelism and in some cases such as monotonicity the properties can be derived automatically [75]–[78].

The SPF uses uninterpreted functions at compile time to represent the memory accesses, loop bounds, and reordering functions that will not be made explicit until runtime. The key extensions to the compiler abstractions that the SPF provides can be summarized as follows.

- Nonaffine iteration spaces: We use uninterpreted functions, functions that are unknown at compile time and determined at runtime. For example, the set that represents the loop over rows and nonzeros per row for a compressed sparse row (CSR) representation in Fig. 1 would be \{ (i, k) | 0 \leq i < N and rowptr(i) \leq k < rowptr(i+1) \}.

- Nonaffine data accesses: We also use uninterpreted functions to specify data access information for sequences of loops that are represented in a loop chain abstraction [53]. For example, the data access function for the \(A[i, k] \rightarrow \{ u[i] | u = col(k) \} \) read in Fig. 1 would be \{ (i, k) \rightarrow \{ u[i] | u = col(k) \} \}.

- Nonaffine transformations: In the compiler, we specify the information that will be passed from the inspector to the executor using uninterpreted functions. For example, the coalesce transformation in [73] and shown in Fig. 4 creates a new scheduling function to map the 2-D nonaffine, SpMV iteration space to 1-D affine one, \( T_{\text{coalesce}} = \{ (i, k) \rightarrow (k') | k' = c(i, k) \wedge 0 \leq k' < NNZ \} \), where \( c \) and \( NNZ \) are uninterpreted functions at compile time. At runtime, \( c \) explicitly maps iterations of the 2-D CSR SpMV loop into a 1-D loop.

As was described in Section III-A, for code generation it is necessary to solve for the new iterators in terms of the old iterators. This needs to be possible in SPF as well. For example, it is important for the coalesce example in Fig. 4 that the compiler knows the \( c() \) function that maps iterations from the original 2-D iteration space to a 1-D iteration space is invertible. The inverse \( c_{\text{inv}}() \) is used to convert the new 1-D iterator back to the old iterators in the updated array accesses.

For correctness, transformations should also satisfy the data dependences. Showing correctness of the transformations in SPF is more difficult due to the uninterpreted functions becoming explicit at runtime. Therefore, showing correctness at compile time means reasoning about the inspector implementations. Some preliminary work has been done to show how correctness can be determined at compile time using mechanized proof by Norrish and Strout [16].

\(^3\)https://github.com/CompOpt4Apps/IEGenLib

### Original Iteration Space

\[
I = \{ (i, k) | 0 \leq i < n \wedge rowptr(i) \leq k < rowptr(i+1) \}
\]

#### Original SpMV code

```c
for (i=0; i<n; i++) {
    for (k=rowptr[i]; k<rowptr[i+1]; k++) {
        y[i] += a[k]*x[col[k]];
    }
}
```

#### Transformation: coalesce

\[
T = \{ (i, k) \rightarrow (k') | k' = c(i, k) \wedge 0 \leq k' < NNZ \}
\]

\[
\text{NNZ} = \text{count}(I)
\]

\[
c = \text{order}(I)
\]

#### Inspector code

```c
NNZ = \text{count}(\text{rowptr}, n);
c = \text{order}(\text{rowptr}, n);
c_{\text{inv}} = \text{inverse}(c);
```

#### Executor code

```c
for (k'=0; k'<\text{NNZ}; k'++) {
    y[c_{\text{inv}[k']}[0]] += a[c_{\text{inv}[k'][1]]]*x[col[c_{\text{inv}[k'][1]]]];
}
```

---

**Fig. 4.** The sparse coalesce transformation presented in [73] can convert a loop with any depth of loop nesting to a 1-D loop with affine bounds. The inspector counts the number of iterations in the original loop (denoted as iteration space \(T_I\)) to determine an upper bound, and maintains a mapping from this count to the original loop iteration values that are used to access the arrays. The transformation specification for coalesce includes an indication of how the uninterpreted functions \(c\) and \(c_{\text{inv}}\) are to be constructed using the iteration space \(T\) as input. The actual code that calls \(\text{count()}\) and \(\text{order()}\) sends the variables that describe \(T\), specifically \(\text{rowptr}\) and \(n\), as arguments. The \(\text{order()}\) function above iterates through the original iteration space and stores a mapping of the 2-D iterations \((i, k)\) to a 1-D count of the iterations into \(c\). See [73] for further optimizations that are possible to remove unnecessary indirect accesses in the executor code.

In summary, the use of uninterpreted functions in SPF enables representing sparse computations, transforming those computations, composing such transformations, code generation, and determining correctness. Since there is a framework, it is also possible to develop scheduling algorithms for automatically determining compositions of transformations. Developing such scheduling algorithms is future work.

### C. Applicability of Composition in SPF

By using uninterpreted functions, SPF is able to express most of the inspector–executor transformations that have been developed and compose them with each other and with polyhedral loop transformations. The data and iteration reordering example in Section IV-B illustrates some of the data locality optimizations that can be expressed in SPF. Those transformations are 1-D, but multidimensional transformations such as tiling are also possible. Sparse tiling transformations have been shown to be expressible in SPF [30], [31] and so has converting between sparse matrix formats such as CSR to blocked CSR [28].
Wavefront parallelization is expressible in SPF, although as with the polyhedral model it does require indicating which dimension of the transformed iteration space should be parallelized. Parameters such as which loop should be parallelized and tile size are provided by the user in the transformation script or derived automatically by the compiler (see Fig. 2).

Moving beyond shared memory parallelization to distributed memory parallelization is possible. Distributed memory parallelization consists of inspectors that determine communication schedules. The communication schedules map a pair of process identifiers to a set of data locations that need to be communicated from the first process to the second. This schedule can be represented as an uninterpreted function at compile time and an explicit function at runtime. A challenge with distributed memory parallelization is proper derivation and insertion of the message passing calls in the appropriate locations in the transformed code. There is promise that SPF can be extended to do this due to recent progress generating distributed memory implementations with the polyhedral model [79]–[81].

In general, the SPF enables mapping an original iteration and data space to a new iteration and data space. Once one or more of the dimensions in the transformed iteration space has been indicated as parallel, there are many different approaches to implementing that parallelization: kernel on an accelerator, MPI parallelization, OpenMP parallel loops, a tasking programming model, etc. Various techniques have been developed to optimize the parallel implementations on various architectures. For example, Venkat et al. [28] developed techniques for vectorizing executors on GPUs [28] and other work looks at minimizing the synchronization needed on multicore architectures when implementing wavefront parallelization [15]. Determining which implementation would work best might require an inspector to determine information such as the size of the parallel loop.

D. Power of SPF to Enable Code Optimization

The SPF uses uninterpreted function symbols to represent index arrays whose values will not be known until runtime. However, the use of the term uninterpreted is somewhat of a misnomer. The values for the index arrays are not known at compile time, but their domain and range, various properties such as monotonicity and bijectivity, and some of their relationships with other index arrays are often known. Using this information enables compile-time simplifications. This section provides some examples of the kinds of simplifications that are possible.

1) Simplifying Array Accesses: As with the polyhedral model, SPF maintains each data array access as a separate function. Keeping track of the memory access functions separately and knowing if an uninterpreted function is going to be bijective enables simplifications in some cases. For example, if a data array is reordered using a bijective function \( \delta \) and the loop that is accessing the array is also reordered with \( \delta \), then the access to the data array will become \( \delta(\delta^{-1}(k')) \) in the executor. This can clearly be simplified to \( k' \) therefore removing unnecessary indirection in the transformed executor code. In addition to compile-time simplification, it is possible to apply inspector-executor simplifications such as pointer update [20], [31], which has an inspector collapse nested index arrays into a single index array.

2) Reducing Inspector Loop Depth: Some inspectors such as those for full sparse tiling and wavefront need to iterate over the data dependences between various iterations of a loop. The data dependences can be set up as a set with constraints and then a loop nest in the inspector can iterate over the variables in the set and determine when the conditions are satisfied and thus a dependence exists.

In [29], we showed that by using information such as the monotonicity of some of the index arrays (e.g., the values in \texttt{rowptr} in Fig. 1 is monotonically nondecreasing) and some of the relationships between index arrays (e.g., \texttt{rowptr}[x] \leq \texttt{diagptr}[x]), it is possible to determine some of the data dependence set variables as functions of other set variables. This enables removing a loop nesting depth from the generated inspector and thus significantly improving the performance of the inspector.

IV. INSPECTOR CODE SPECIFICATION AND COMPOSITION

The compile-time specification of inspector-executor transformations also includes details on how those uninterpreted functions will become explicit at runtime. Those inspector specifications and the needs of the transformed executor code are then converted into an inspector dependence graph (IDG) for the generation of composed inspector code.

A. Inspector Specification

The inspector code (see Fig. 4 for an example) inspects explicit sets and functions that represent the runtime instantiations of things such as the iteration space, data mappings, and data dependences. Thus, we have the indication that \( \text{NNZ} = \text{count}(\text{I}) \) and \( c = \text{order}(\text{I}) \) in the coalesce transformation specification in Fig. 4, where \( \text{I} \) is the integer tuple set for the original loop and \text{count} and \text{order} built-in inspector routines. The \text{count} routine will at runtime count all of the iterations in the given integer tuple set. The \text{order} routine will map each iteration in the given integer tuple set to a 1-D lexicographical order (e.g., \( c(0,0) = 0, c(0,1) = 1, c(1,2) = 2, c(2,3) = 3, \ldots \)).

The input explicit sets and functions are parameters and index arrays such as \( \text{n}, \text{col}, \) and \text{rowptr} in the original SpMV code of Fig. 4. The outputs of inspector code are the new uninterpreted functions being introduced by the specified transformation. In the Fig. 4 example,
The uninterpreted functions NNZ, c, and c_inv are made explicit by the inspector.

Since the compiler is now generating inspector code in addition to the transformed executor, we introduce a representation of the various inspector components and indicate how they share information with each other with an IDG. Fig. 5 illustrates a high-level representation of the IDG for the coalesce inspector example shown in Fig. 4. Each rectangular node indicates an explicit function, which is the runtime instantiation of an uninterpreted function. Each elliptical node represents a compiler-generated, runtime library, or programmer-provided inspector function.

B. Composing Inspector Specifications

The SPF enables the composition of multiple inspector-executor transformations. The output from one inspector can be the input to another inspector. This approach also enables hiding the algorithms that perform graph partitioning, data and iteration reordering heuristics, etc., behind their output, which will be uninterpreted functions. Inspector composition is the connection of outputs from one inspector to the inputs of another. Transformation specified in Fig. 4.

Fig. 5. Inspector Dependence Graph (IDG) for coalesce transformation specified in Fig. 4.

The key thing to note about the composition example in Fig. 6 is that the input to the iteration reordering heuristic is the updated data access to the newly reordered y’ array, A_{k → y’}. Fig. 7 shows how the inspector IDGs are composed. This is how composition of inspector-executor transformations occur. Later transformations will be using the data access functions, data dependences, and iteration spaces that have been modified by previous transformations. The uninterpreted functions enable the compiler to represent those changes at compile time.

Fig. 6 shows the changes to the computation and data specification after each transformation is applied to illustrate the effect of the first transformation. The transformations from the original code to the executor can all be composed at compile time with only the final version of the transformed code being generated.

V. FUTURE DIRECTIONS: GENERALIZING INSPECTOR-EXECUTOR OPTIMIZATIONS

We are currently developing a compilation system, CHiLL-I/E (pronounced like the delicious food), that combines polyhedral code generation and inspector—
This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

---

**Fig. 6.** An example that illustrates composition of transformations by showing the composition of data and iteration reorderings applied to a COO version of SpMV. The original code is in the top left rectangle. To the right of that rectangle is the SPF set and relation representations of the data spaces, iteration space, and data access functions. The transformation specifications are shown next to two large arrows. Note that the mathematical specification for the data reordering takes the access function \( AK \rightarrow Y \) as input, whereas the inspector code needs the \( NNZ \) and \( row[] \) variables for the data reordering heuristic to actually traverse that access function. After each transformation is applied, the code changes and the representations of portions of code and data representation change.

---

The combination will enable efficient composition of affine and nonaffine loop transformations. Efficient composition requires specializing the data structures communicated between inspectors and between inspectors and executors as well as applying loop optimizations to the inspectors, for instance fusing two inspectors that iterate over the same data.

Fig. 2 illustrates the design of the CHiLL-I/E compilation system. CHiLL-I/E will compose a series of generated or hand-written inspectors and generate an executor to replace the original sparse calculation. The inspectors will share information with each other and with the executor using an explicit function data structure. The CHiLL system already includes an autotuning framework, which we will utilize to guide transformation sequence selection and parameterization.

---

### A. CHiLL: Compiler and Autotuning Framework

The CHiLL compiler and autotuning framework provides a high-level interface, separate from primary source code, to apply execution schedule transformations. The transformations are expressed as a script (referred to as a recipe) that is applied to sections of C or C++ source code. CHiLL recipes provide a mechanism to transform the application code and perform autotuning without negatively impacting the maintainability of the original code. The scripts are either hand-written or derived by a decision algorithm within CHiLL [82]; each identifies the location of the original source code along with a series of loop transformations, e.g., shift, fuse, and skew. The transformations are composed and applied, and new code is generated.

CHiLL utilizes polyhedral code generation to perform transformations. Additionally, CHiLL supports a set of nonaffine transformations: coalesce, compact, compact and pad, split, and make dense [28], [73]. The implementations of each of these transformations has been optimized by specializing the code generation for the specific transformation. In CHiLL-I/E, the plan is to provide an API for explicit functions and specialize the explicit function implementations based on their characteristics and usage through that API. Code generation that accommodates uninterpreted functions has been done for specific sequences of transformations, but those algorithms will be generalized.

### B. I/E API and Common Component Library

An Inspector–Executor API will communicate semantics utilized in optimizations to the compiler framework. There are three primary components to the API and
library: a common input and output interface for each inspector (explicit functions), a predefined set of explicit function iterators, and a common set of well-understood inspectors that can be optimized by the compiler.

Explicit functions are an abstraction to represent the uninterpreted functions from SPF’s compiletime at runtime. They are more general than index arrays, which tend to represent 1-D to 1-D mappings. Explicit functions can represent the mapping of \( k \)-dimensional integer tuples to \( v \)-dimensional integer tuples. The data structures used to store an explicit function is generated by the compiler and can be specialized per inspector or set of inspectors. The inspector generated to apply a coalesce transformation for the SpMV example in Fig. 4 demonstrates this. The input dimensionality for the inverse coalesce function is 1-D. The output dimensionality is 2-D and each input will only have one tuple in its output set. Based on the input and output dimensionality of the explicit function it is implemented as an array of structs, where each struct has two fields. The research that introduced the coalesce transformation performed this specialization specific to the coalesce transformation. The concept can be generalized.

Limiting the input and output of inspectors to be explicit functions also allows the compiler to create an IDG. IDGs are essentially data flow graphs for inspectors and inform the optimizations discussed in the next section.

Most inspectors iterate over the input explicit functions collecting information. High-level knowledge of the iteration order and data dependencies dictating that order allows the compiler to either reorder the iterations in a legal manner or construct the explicit functions in a manner that improves the memory access patterns.

There are some inspectors that are common across many I/E transformation. Examples include count and invert. A collection of common inspectors will be provided in a library. Inspectors with clear semantics, such as count, can be added to any handwritten inspector that uses the provided iterators.

The I/E API and library are designed to support a set of commonly used optimizations on the inspectors and executors themselves. The following section describes some of the potential optimizations and how they depend on the components of the proposed API and library.

C. Generalizing Inspector Optimizations

Hand-written inspectors and executors outperform those that are compiler generated. The reason is that expert programmers are able to identify and exploit optimization opportunities. Given a series of inspectors and adequate semantic information the compiler may be able to apply optimizations to that series of inspectors in a generic way. The opportunities for optimizations include: specializing the explicit function, both by reducing space requirements and adjusting the data layout of the explicit functions to match the iteration order of the associated inspectors; fusing inspectors that are iterating over common data; fusing inspectors that have a producer-consumer relationship with respect to the explicit functions; and identifying opportunities for parallelization. There will often be trade-offs between these choices and autotuning can be used to automate those decisions. The following are examples of fusion and the semantic knowledge required to apply it.

1) Fusing Inspectors: Fusing some inspectors reduces the number of times that the same data is iterated over and for other inspectors it may avoid the need for temporary data. Opportunities of fusion are identified by examining the IDG. This representation makes it possible to determine discover possible fusion at compile time.

The IDG shows clearly which explicit relations will be passed into an inspector routine as input. If two inspectors operate on the same input, it is possible that loops of the inspectors can be fused. Handwritten inspectors and/or those in a library would need to be fused by hand, but automatically generated inspectors could be fused before the compiler generates them.

The IDG also clearly identifies producer-consumer relationships between inspectors. In this case fusion may prevent the use of temporary storage. The coalesce example in Fig. 5 demonstrates both of these cases.

In [27], the CHiLL compiler was extended to generate a coalesce inspector that determined the mapping of the 2-D irregular iteration space to a continuous 1-D iteration space, order in Fig. 5. At the same time it computed the number of iterations that would be needed in the 1-D iteration space, count. In the IDG, this can be shown as the zero-dimensional uninterpreted function NNZ.
being computed as the cardinality of the original iteration space, and the coalesced uninterpreted function \( c(i, j) \) being computed as a total ordering on the original iteration space. They both need to be computed and the inspectors that compute them will require iterating over the original iteration space loops to compute them. Additionally, the inspector invert consumes the explicit function produced by order. All three inspectors can be fused and calculated simultaneously. The space required by order is eliminated because its inverse is generated directly.

In [27], the fusion was hardcoded in the inspector code generator, however, if such portions of inspectors are explicitly expressed in a separate fashion, it provides the opportunity for automated fusion among not only these two inspector computations but others if additional inspector–executor transformations are to be composed.

2) Inspector Parallelization: In [29] a parallel executor was generated by identifying level sets in a data flow diagram and implementing wavefront parallelism. Additionally, the inspector was parallelized. This was achieved by specializing the explicit function to avoid data dependences across iterations of the outer loop in the inspector. Instead of always storing the dependences at the source iteration in an adjacency matrix data structure, the incoming and outgoing dependences for a particular iteration are stored together thus avoiding conflicts between inspector outer loop iterations. Generalizing this optimization requires both specialization of the explicit functions and knowledge of the iteration patterns.

VI. CONCLUSION

Inspector–executor techniques bridge the gap between static optimization in a compiler and runtime optimization and scheduling. In this paper, we survey research about the incorporation of inspector–executor transformations into compilers, and we articulate the need for general techniques to generate and compose inspectors, even existing inspectors written manually. We describe the role of the SPF in filling that need, by providing high-level abstractions and the IDG to reason about the dependences across and within inspectors; this information allows the compiler to reason about optimization opportunities such as parallelization and fusion of inspectors. We describe our future work on CHiLL-I/E, a system we are developing to leverage the polyhedral transformation and code generation capabilities of CHiLL with the sophisticated dependence simplification and generalized inspector generation. Indeed, the ability to compose inspector–executor transformations with other compiler transformations fills a significant role in the march toward sophisticated compiler technology for today’s complex architectures.

Acknowledgments

The authors would like to thank the anonymous reviewers for their constructive comments.

REFERENCES

[27] A. Venkat, M. Hall, and M. M. Stout, “Loop and data


AUTHOR QUERIES

AUTHOR PLEASE ANSWER ALL QUERIES

PLEASE NOTE: We cannot accept new source files as corrections for your paper. If possible, please annotate the PDF proof we have sent you with your corrections and upload it via the Author Gateway. Alternatively, you may send us your corrections in list format. You may also upload revised graphics via the Author Gateway.

AQ:1 = Please provide photos/bios for both authors.