Bridging the Gap Between General-Purpose and Domain-Specific Compilers with Synthesis

Alvin Cheung[1], **Shoaib Kamil**[2], **and Armando Solar-Lezama**[2]

1 **University of Washington, US**
2 **MIT CSAIL, US**

Abstract

This paper describes a new approach to program optimization that allows general purpose code to benefit from the optimization power of domain-specific compilers. The key to this approach is a synthesis-based technique to raise the level of abstraction of general-purpose code to enable aggressive domain-specific optimizations.

We have been implementing this approach in an extensible system called Herd. The system is designed around a collection of parameterized *kernel translators*. Each kernel translator is associated with a domain-specific compiler, and the role of each kernel translator is to scan the input code in search of code fragments that can be optimized by the domain-specific compiler embedded within each kernel translator. By leveraging general synthesis technology, it is possible to have a generic kernel translator that can be specialized by compiler developers for each domain-specific compiler, making it easy to build new domain knowledge into the overall system.

We illustrate this new approach to build optimizing compilers in two different domains, and highlight research challenges that need to be addressed in order to achieve the ultimate vision.

1998 ACM Subject Classification I.2.2 Program Synthesis

Keywords and phrases compilers, domain-specific languages, program synthesis, cross compilation, verification

Digital Object Identifier 10.4230/LIPIcs.SNAPL.2015.51

1 Introduction

Despite significant advances in compiler optimization over the last thirty years, the promise of actual optimality in generated code remains elusive; even today, clean and high-level code cannot compete with heavily optimized hand-tuned, low-level code, even if the former is compiled with the best optimizing compilers. Part of the problem is that the kind of aggressive, high-level, global optimization that a performance engineer can perform by hand often requires domain knowledge about the implementation strategies that are best for particular classes of programs. For a general-purpose compiler, it is often difficult to tell that a piece of code comes from a particular domain, and it is also difficult to incorporate the requisite domain knowledge in a traditional compiler architecture.

Domain-specific languages (DSLs) have emerged as a solution to the challenge of obtaining high performance from high-level code. Compilers for domain-specific languages leverage specialized internal representations and domain-specific transformations to encode the domain knowledge necessary to optimize a given class of programs. In a number of different domains [32, 24, 29, 23, 4], it has been shown that domain-specific compilers can outperform traditional compilers – and in some cases can even surpass hand-tuned code created by performance experts – because they can combine the domain knowledge embedded in them with exhaustive exploration to produce truly optimal implementations.

While DSLs ease the job of the compiler developer in generating efficient programs, they often come at a cost for application developers. First, given the proliferation of DSLs even within a given domain, the developers must understand the pros and cons of each DSL in order to choose the best one for their applications. Once chosen, developers might need to rewrite their existing applications into the DSL. Worse yet, developers might only want to express fragments of their computation in the DSL. As a result, the program might become a collage of code fragments written in various DSLs – database backed applications, where for years it has been customary to combine a general-purpose language like Java with query languages like SQL, provide a cautionary tale of the software engineering challenges inherent in this approach [19, 5].

In this paper, we present a system called HERD that we are building for bridging the gap between general-purpose compilers and DSLs. Given a program written in a general-purpose language, our system relies on a new technology called *Synthesis Enabled Translation* (SET) which systematically identifies code fragments that can be translated into DSLs. SET derives provably correct translations for each identified code fragment, and the translations are then given to domain-specific compilers for aggressive optimizations. We have demonstrated SET on two domains for which efficient domain-specific notations exist: stencil computations and database queries, but the generality of the SET approach should make it possible to extend the system to other domains for which high-performance DSLs are available.

With SET, compiler developers can add support for a new DSL by describing the semantics of the target language, together with high-level filtering rules to help guide the compiler towards more promising code fragments. The compiler can then leverage recent advances in program synthesis and software verification techniques [6, 13] to dynamically search for code fragments that can be rewritten into the target DSL. Using the provided specification of the target DSL, the compiler will automatically weave the generated DSL code back into the original program in order to complete the optimization process.

Overall, our approach has the following advantages over both general-purpose compilers and direct use of DSLs:

- Compared to direct use of DSLs, our approach does not require programmers to learn new formalisms, or even to know about the existence of DSLs for particular domains.
- The translations to DSLs performed by our system are provably correct, so there is less risk of introducing bugs in the program through the translation process.
- The compiler can automatically search through transformations across multiple DSLs simultaneously given an input program, and rewrite appropriate fragments into the optimal DSL according to a cost metric. This eliminates the application developers' need to rewrite their code into multiple DSLs and maintain each implementation.

In the rest of this paper, we first briefly describe constraint-based synthesis in Sec. 2 and outline the proposed architecture of HERD in Sec. 3. Then, in Sec. 4 and Sec. 5 we describe instances where we have applied SET in constructing compilers for two very different application domains, namely, applications that interact with databases for persistent storage and stencil computations. The HERD architecture is designed after our experience from the two application domains. After that, we discuss related work in Sec. 6 and conclude in Sec. 7.

2 Constraint-Based Synthesis

In this section we describe the synthesis technology that forms the basis of SET. SET uses constraint-based program synthesis [28, 1], which provides mechanisms to efficiently search a space of candidate implementations for one that satisfies a given functional specification.

Specifically, the synthesis problem is formulated as solving the Doubly Quantified Boolean Formula problem (2QBF):

$$\exists c. \forall x. Q(f_c, x)$$

Here f_c represents the unknown function parameterized by a set of control values c. Given c, the predicate Q checks to see whether f_c satisfies all the constraints under all possible inputs x. The job of the synthesizer is to find such c. Since the number of possible inputs is infinite, in practice, synthesizers perform bounded verification where the size of x is limited. For instance, if the inputs are lists, then one way to perform bounded verification is to limit the maximum number of elements in each list, or the size of the elements in each list. If the inputs are integers, then bounded verification can be done by checking constraint satisfaction for all integers up to a certain value.

In SET, we use synthesis to discover the DSL program and any program invariants that are necessary to prove equivalence to the original program. Thus, the constraints in Q consist of clauses that ensure equivalence to the semantic meaning of the original snippet.

Unlike typical compiler analysis techniques, which use semantics-preserving rewrites to transform the input code to the target language in a deductive way, the synthesis technology in SET uses inductive reasoning: it works by searching for solutions that satisfy the specification in concrete scenarios (i.e., pairs of program inputs and outputs), and then checks whether the solution generalizes for all possible program inputs. This approach is formalized in an algorithm called CEGIS [28]. CEGIS avoids the universal quantifier from the 2QBF problem shown above when searching for c by instead solving a simpler problem: given a small set of representative inputs X_i, it attempts to find c_i such that

$$\forall in \in X_i . Q(f_c^i, in)$$

CEGIS constructs the set of inputs X_i in an iterative manner. First, X_0 is constructed from a single random input, and CEGIS uses a constraint solver to solve the problem to obtain c_0 (and the resulting f_c^0). These control values are then passed to a checking procedure that checks whether the resulting program is correct for all inputs. If so, then we have found c. Otherwise, the checking procedure generates a counterexample input in_0. in_0 is then added to X_0 to construct X_1, and the process is repeated until a c is found that generalizes to all possible inputs. To speed up the search process, the search for c_i during each iteration is done in a symbolic manner: rather than enumerating and checking each c one by one, CEGIS encodes the parameterized program Q and as a constraint system, with f_c as the solution. Hence during each iteration solving the constraint system will identify the next f_c^i to use.

3 Herd Architecture

We now describe the overall architecture of HERD based on SET methodology, as shown in Fig. 1. After parsing the input code, HERD passes the AST to a series of static program analyzers to annotate the program with type, aliasing, and other structural information about the input. The language broker then takes the AST, along with analysis results, and sends them to the kernel translators registered with HERD. Each kernel translator represents a target DSL. The goal of each translator is to identify and translate code fragments from the input AST into the target DSL, and return the translated code fragment to the fragment reassembler. After receiving the translated code fragments from each of the kernel translators, the reassembler combines them to produce the final executable, potentially using a cost model to evaluate among different options in case the same code fragment has been translated by

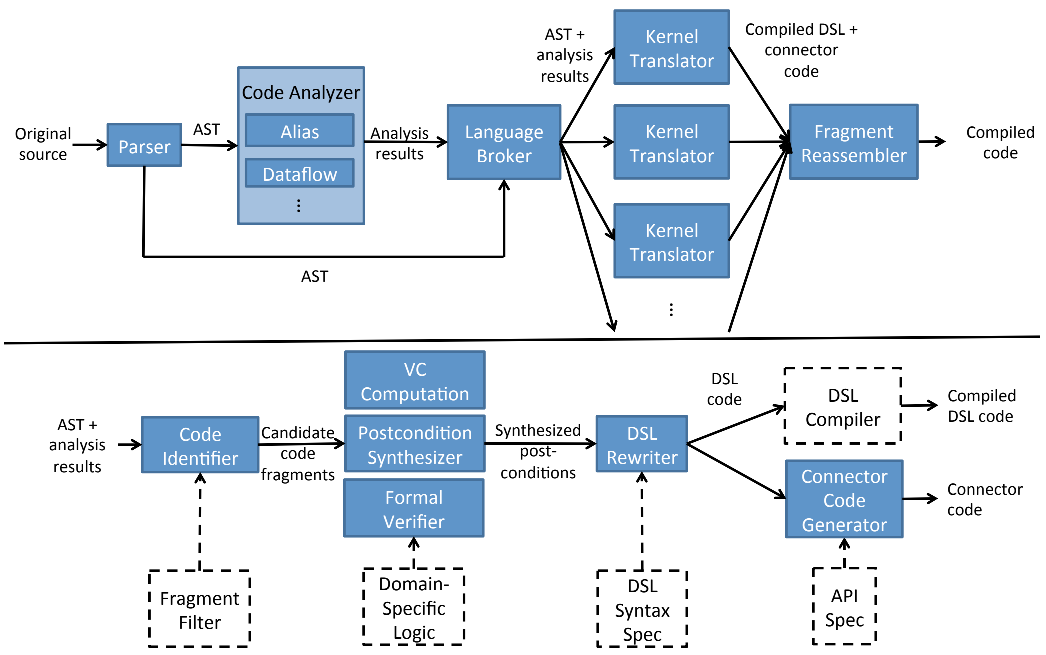

Figure 1 (a) Architecture of HERD (top); (b) kernel translator detail (bottom), with dotted components representing inputs from the DSL compiler developer when constructing a new kernel translator.

multiple translators. In the rest of this section, we describe the details of kernel translators and how compiler maintainers can use HERD to generate them for new target DSLs.

3.1 Kernel Translators

Kernel translators take in general-purpose ASTs and attempt to transform portions of the input into the given DSL. The translation consists of the steps described below.

3.1.1 Label potential code fragments

The first step in DSL translation is to identify candidate code fragments from the input AST that might be amenable to translation. To aid in fragment identification, the compiler maintainer provides HERD with filtering conditions. For instance, to compile for a DSL that targets parallel computation, the developer might specify all loop nests with array writes as candidates. Examples of such hints are discussed in Sec. 4 and Sec. 5. Using such conditions, the HERD code identifier automatically labels a number of candidate code fragments from the input program during compilation, and passes them to the next component in the kernel translator.

3.1.2 Search for rewrites

For each candidate code fragment that the code identifier found, the kernel translator then tries to express it in the target DSL. The search for semantically-equivalent constructs in the target DSL is framed as a synthesis problem, with the goal to find DSL expressions that can be proved to be semantically equivalent to the original code using Hoare-style program verification. This is done in two steps. First, compiler maintainers provide HERD with a high-level language that abstracts and formalizes the target DSL. The postcondition synthesizer then takes the formalized DSL and tries to find postcondition expressions written using that language. The search space for each postcondition expression is limited to only non-trivial expressions (i.e., no expression such as "x = x"). In addition to the postcondition, the synthesizer also searches for invariants that are needed to verify loop constructs in the candidate code fragment. However, unlike prior work in inferring loop invariants, we only need to find loop invariants that are strong enough to establish the validity of the postcondition.

3.1.3 Verify and convert

To reduce the time needed to synthesize postconditions, the kernel translator uses a bounded synthesis procedure as described in Sec. 2, where it checks validity of the found postcondition up to a fixed heap size, and only expressions up to a fixed length are considered. Other search techniques can be used as well. Due to bounded reasoning, any candidate postcondition that is discovered by the synthesizer needs to be formally verified. If the candidate fails formal verification, the kernel translator will ask the synthesizer to generate another postcondition candidate (by increasing the space of expressions considered, for instance). Otherwise, the DSL rewriter translates the postcondition into the target DSL using the syntax specification provided by the compiler maintainer.

3.1.4 Generate connector code

The translated DSL code is then compiled using the DSL compiler provided by the maintainer. At the same time, the connector code generator produces any wrapper code that is necessary to invoke the compiled DSL code fragment, for instance passing parameters and return values between the general-purpose and DSL runtimes. The DSL compiler maintainer provides an API specification of the DSL runtime to the system to facilitate the generation of wrapper code.

The final output of the kernel translator consists of the compiled DSL code along with any wrapper code, both of which are returned to the fragment reassembler. As mentioned earlier, the fragment reassembler replaces the original source code AST with translated DSL code fragments. In some cases, multiple kernel translators can find rewrites for a given code fragment. For instance, a code fragment that can be converted into SQL may also be expressible in Hadoop for parallel processing. In such cases, the reassembler chooses the translation that is most optimal. To estimate the cost of each translation, HERD executes each translation using input test cases, generating random test data if necessary. The reassembler then chooses the translation with the lowest cost (e.g., with the lowest expected execution time) and replaces the input source code with most optimal translation. HERD can use other mechanisms to estimate cost and choose the optimal translation as well (e.g., using auto-tuning [2]), and a runtime component can also be added to adaptively choose the optimal translation based on program input, and we leave that as future work.

In the next sections, we highlight two application domains where we have used the SET methodology to build DSL compilers. We constructed kernel translators for each domain to convert fragments of general-purpose code into different DSLs, and the converted code fragments achieve up to multiple orders of magnitude performance speedup compared to the original general-purpose code by utilizing domain-specific optimizations. Without translating to DSL, it would be very difficult for the original program to utilize the optimizations provided by the DSL runtime.

4 Transforming Java Code Fragments to SQL

In this section we describe our experience in building a kernel translator to enable general-purpose code to utilize specialized data retrieval functionality provided by database engines in the context of QBS (Query By Synthesis) [9]. QBS takes in Java code fragments that compute on persistently-stored data, and automatically converts them into database queries (SQL in this case).

4.1 Background

Many applications make use of database systems for data persistence. Database systems typically provide an API for applications to issue queries written in a query language such as SQL, while application frameworks [12, 26, 15, 21] encourage developers to modularize data persistence operations into libraries. This results in an intermixing of database queries and application code, making code maintenance more difficult. Moreover, as implementors of persistent data libraries cannot anticipate all possible ways that persistent data will be used in the application, this leads to application developers writing custom data manipulation code in the application rather than making use of the highly-optimized implementations provided by database engines, potentially resulting in substantial application slowdown. The goal of QBS is to address such issues by converting general-purpose application code that computes on persistent data into semantically-equivalent queries in SQL.

4.2 Kernel Translator

QBS was built as a kernel translator registered with HERD. In this section we describe the different inputs used to construct the kernel translator.

Fragment filtering. QBS considers all code fragments that can potentially use persistent data as candidate fragments for translation. To identify such code fragments, QBS first finds all statements that retrieve data from persistent storage (by identifying standard API calls [17]) and have no side-effects (since those have no semantic equivalence in standard SQL). An interprocedural taint analysis is then used to identify all statements that might take in persistent data as input, and a greedy algorithm is used to group together such statements to form candidate code fragments.

Domain-specific Logic. QBS defines its formalized DSL as shown in Fig. 2(a). The language is an imperative one that includes a number of operators derived from relational algebra (e.g., selection, projection, etc), except that they operate on ordered collections rather than relations, since the Java data persistence API (like many other popular ones) is based on ordered collections. The semantics are defined using axioms based on the theory of lists, a sample of which are shown in Fig. 2(b). Given this DSL, QBS makes use of the components included in the kernel translator to search for possible rewrites for each candidate code fragment, as described in Sec. 3.1.

Output DSL Syntax. The formalized DSL used by QBS was designed to enable easy translation to the target domain language (SQL). As such, QBS defines syntax-driven rules to transform postcondition expressions into appropriate SQL queries, the details of which are discussed in [9].

$$
\begin{array}{rcl}
\mathsf{c} \in \mathsf{lit} & ::= & \mathsf{True} \mid \mathsf{False} \\
 & \mid & \mathsf{number\ literal} \mid \mathsf{string\ literal} \\
\mathsf{e} \in \mathsf{exp} & ::= & \mathsf{c} \mid \mathsf{program\ var} \\
 & \mid & \{\mathsf{f_i} = \mathsf{e_i}\} \mid \mathsf{e_1\ op\ e_2} \mid \neg\ \mathsf{e} \\
 & \mid & [\ ] \mid \mathsf{Query}(\ldots) \\
 & \mid & \mathsf{size(e)} \mid \mathsf{get(e_r, e_s)} \\
 & \mid & \mathsf{top(e_r, e_s)} \mid \pi(\mathsf{e}, \mathsf{f}_\pi) \mid \sigma(\mathsf{e}, \mathsf{f}_\sigma) \\
 & \mid & \bowtie(\mathsf{e_1}, \mathsf{e_2}, \mathsf{f}_\bowtie) \mid \mathsf{append(e_1, e_2)} \\
 & \mid & \mathsf{sort(e, [e.f_i])} \mid \mathsf{unique(e)} \\
 & \mid & \mathsf{sum(e)} \mid \mathsf{max(e)} \mid \mathsf{min(e)} \\
\mathsf{f}_\pi(\mathsf{e}) & ::= & \{\mathsf{e.f_i}\} \\
\mathsf{f}_\sigma(\mathsf{e_s}) & ::= & \wedge\ \mathsf{e.f_i\ op\ c} \mid \mathsf{e.f_i\ op\ e.f_j} \\
 & \mid & \mathsf{contains(e_s, e)} \\
\mathsf{f}_\bowtie(\mathsf{e_1}, \mathsf{e_2}) & ::= & \wedge\ \mathsf{e_1.f_i\ op\ e_2.f_j} \\
\mathsf{op} \in \mathsf{bop} & ::= & \wedge \mid \vee \mid > \mid =
\end{array}
$$

$$\boxed{\text{axioms for top operator}}$$

$$\mathsf{top}([\ ], \mathsf{i}) = [\ ]$$

$$\mathsf{i} = 0 \to \mathsf{top}(\mathsf{r}, \mathsf{i}) = [\ ]$$

$$\mathsf{i} > 0 \to \mathsf{top}(\mathsf{h} : \mathsf{t}, \mathsf{i}) = \mathsf{h} : \mathsf{top}(\mathsf{t}, \mathsf{i} - 1)$$

$$\boxed{\text{axioms for projection } (\pi)}$$

$$\pi([\ ], \mathsf{f}) = [\ ] \qquad \pi(\mathsf{h} : \mathsf{t}, \mathsf{f}) = \mathsf{f}(\mathsf{h}) : \pi(\mathsf{t}, \mathsf{f})$$

$$\boxed{\text{axioms for selection } (\sigma)}$$

$$\sigma([\ ], \mathsf{f}) = [\ ]$$

$$\mathsf{f}(\mathsf{h}) = \mathsf{True} \to \sigma(\mathsf{h} : \mathsf{t}, \mathsf{f}) = \mathsf{h} : \sigma(\mathsf{t}, \mathsf{f})$$

$$\mathsf{f}(\mathsf{h}) = \mathsf{False} \to \sigma(\mathsf{h} : \mathsf{t}, \mathsf{f}) = \sigma(\mathsf{t}, \mathsf{f})$$

Figure 2 (a) Formalized DSL used by QBS (left); (b) Semantic axioms defining the language operators (right).

Connector API Spec. The Java data persistence API defines a number of methods for Java applications to issue SQL queries to the database. QBS makes use of this API to pass the inferred queries to the database. Optionally, the inferred queries can be pre-compiled by the database as stored procedures for efficient execution. In both cases, the kernel translator returns the compiled queries and rewritten code fragments back to the HERD toolchain.

4.3 Results

We used two large-scale open source web applications to evaluate QBS, with the goal of quantifying its ability to convert Java code fragments into SQL. A selection of the results are shown in Fig. 3(a), which shows that QBS can convert a majority of the candidate code fragments. The cases that failed were mostly due to lack of expressivity in the formalized DSL (e.g., the application code uses custom aggregate functions). The maximum amount of time spent in synthesis for any code fragment was 5 minutes, with the largest code fragment consisting of about 200 lines of code.

Fig. 3(b) shows a representative result comparing the execution time containing a converted code fragment. In this code fragment, the original code loops through two lists of persistent data in a nested loop and returns a subset of the objects. QBS converted this code fragment into a SQL join query. By doing so, the converted code can take advantage of the specialized join query implementations in the database system (in this case a hash join based on the object's ID was used), and that resulted in a significant performance gain.

5 Converting Stencil Codes

We now turn to describe our experience in building a kernel translator for converting fragments of serial Fortran that express stencil computations into a DSL called Halide [24]. The resulting system, STNG, generates Halide programs from general-purpose Fortran code. The translated Halide programs compile into vectorized, parallel code that outperforms code produced by general-purpose compilers.

itracker (bug tracking system)		
operation type	# benchmarks	# translated
projection	3	2
selection	3	2
join	1	1
aggregation	9	7
total	**16**	**12**

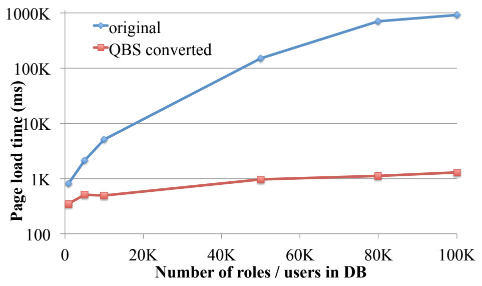

Figure 3 (a) Number of code fragments converted by QBS on one of the applications (left); (b) Performance comparison of converted code fragment (right).

5.1 Background

Stencil computations describe an important computational pattern used in many areas of computing, including image processing, linear algebra solvers, and scientific simulations. A stencil computation consists of updating each point in a multidimensional grid with a function of a subset of its neighbors. Such computations are difficult to optimize using libraries, because the optimizations depend heavily on the sequence of stencils and the particular function being applied. General compiler loop transformations based on the polyhedral method [16] can optimize some stencils, but do not usually obtain peak performance due to their limited scope – they only optimize traversals in an existing loop nest, and they cannot optimize across loop nests due to a lack of high-level knowledge about the overall computation.

A number of domain-specific languages and compilers [29, 18, 10] have been built for stencils, with excellent performance, matching or beating hand-tuned code. We choose to convert stencils to code written in the Halide [24] language, which supports empirical auto-tuning to find best-performing computation schedules for the overall stencil computation.

5.2 Kernel Translator

Fragment Filtering. We search for code fragments that could be stencil computations by examining all loop nests and finding ones that write to arrays, where the indices of the writes are dependent on loop iteration variables. In this process, we also filter out computations and structures that cannot be handled by our domain-specific logic, for instance loop nests that contain side-effect computations.

Domain-Specific Logic. STNG's formalized DSL uses a stylized representation of the space of stencil computations, using the theory of arrays. This representation includes the common constructs found in stencils, including neighbor access, and operations using neighboring points. The major difficulty in synthesis is the complexity of the verification conditions; they include universal quantifiers over arrays, resulting in very large synthesis problems. Our implementation aggressively optimizes the synthesis problem, eliminating symmetries and using domain-specific information where possible to make the problem easier to solve.

Output DSL Syntax. We translate from the formalized DSL to Halide's high-level computation language, which is embedded in C++. In addition, we output a default computation schedule which parallelizes the stencil along the outermost dimension and vectorizes the innermost. We also output utility code that can be used for auto-tuning the schedule.

App	LoC	Found	Translated
StencilMark	245	3	3
CloverLeaf	6635	45	25
NAS MG	1770	9	1

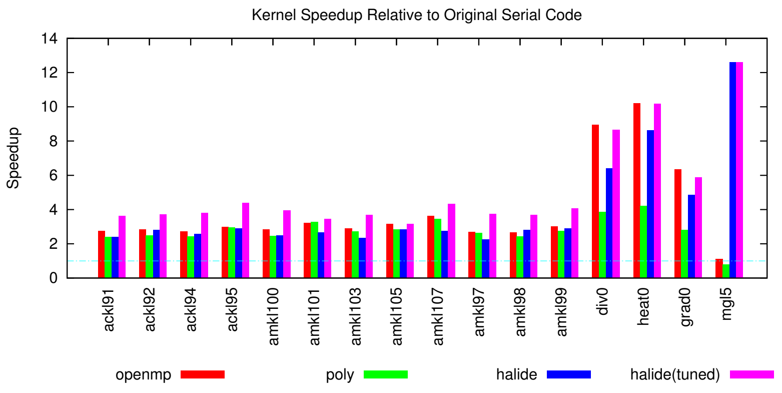

Figure 4 Summary of application results using STNG (top); Performance results (bottom).

Connector API Spec. Connector code is needed to replace the original stencil computation with one using Halide, including creating Halide buffers for each input and output grid. While buffer creation is tedious to write by hand, the automation is fairly straightforward, as it includes converting from Fortran array declarations to Halide buffers that wrap the arrays, and determining the array bounds for each Halide buffer.

5.3 Results

We evaluated STNG on two applications and a set of microbenchmarks for stencil computations, and compare against the state-of-the-art fully automated optimization method using PolyOpt/Fortran [22], a polyhedral compiler for Fortran. Selected results are shown in Fig. 4. As shown in the figure, STNG was able to translate most of the candidate code fragments, and the resulting code outperforms the version that is manually-parallelized or compiled using the polyhedral method.

The STNG prototype has some limitations that we are actively working to address; none of these are due to fundamental issues with the methodology. Currently, we do not support a number of Fortran constructs, including syntax for exponentiation, and do not support conditionals within stencil kernels. These limitations result in only a subset of kernels being translated, as shown in Fig. 4. We are extending the prototype to support these constructs and increase the effectiveness of translation.

6 Related Work

HERD builds on the successes of DSLs in improving application performance across different specialized domains. In this section we discuss other approaches in translating general-purpose to domain-specific code and structuring optimizing compilers.

The usual approach for compiler transformations is deductive: analysis and transformation rules determine which transformations are valid based on deducing characteristics of the original program text. Such approaches are fragile and tiny changes in the program can result in transformation failure [20]. In contrast, SET uses inductive techniques, by searching over combinations of possible constructs to find one with equivalent semantics to the original.

6.1 From General-Purpose to Domain-Specific Code

When converting from general-purpose code to domain-specific code, automated systems derive programmer intent in one of three ways. Pragmas or other annotations (such as those used in OpenMP [11]) require programmers to explicitly denote portions of the code to transform. Alternatively, there has been work on using language features such as types or specific classes [7, 8, 25, 31] for programmers to convey domain-specific information and intent. To our knowledge, Set is the first to use synthesis and verification technology to automatically determine domain-specific intent.

6.2 Code Generation as Search

Traditional optimizing compilers are constructed as a pipeline of code transformations, with each stage making structural changes to the code to improve its performance. Unfortunately, deciding the optimal order to execute each stage is a difficult problem [30]. Recent work has instead focused on using search techniques to find optimized code sequences. For instance, STOKE [27] uses stochastic search to solve the superoptimization problem. Like superoptimization, Herd casts the optimization problem as a search over all possible programs that can be constructed from a number of base operators (constructs in the target DSL in this case). Unlike Herd, however, STOKE performs search over low-level instructions rather than high-level logic expressions.

Meanwhile, inductive synthesis techniques have been used to search for optimal code sequences in data-parallel programs [3] and bitvector manipulation codes [14], although the search was not over multiple potential target languages, as in the case of Herd.

7 Conclusion

In this paper, we describe the Set methodology for building compilers that can automatically convert general-purpose code fragments into DSLs. Furthermore, we constructed Herd to allow DSL designers to use this methodology for building compilers by building kernel translators. We describe our experience applying Herd to build compilers to translate general-purpose code fragments into DSLs (SQL and Halide). A number of challenges remain, such as the limitations of synthesis technologies in finding postcondition expressions, and studying the relative ease of developing kernel translators in comparison to traditional syntax-driven compilers. Future work will extend Herd with new kernel translators, applying the methodology to other application domains.

References

1 Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pages 1–17, 2013.

2 Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. Opentuner: An extensible framework for program autotuning. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, pages 303–316, 2014.

3 Gilles Barthe, Juan Manuel Crespo, Sumit Gulwani, Cesar Kunz, and Mark Marron. From relational verification to simd loop synthesis. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 123–134, 2013.

4 Gerald Baumgartner, Alexander Auer, David E. Bernholdt, Alina Bibireata, Venkatesh Choppella, Daniel Cociorva, Xiaoyang Gao, Robert J. Harrison, So Hirata, Sriram Krishnamoorthy, Sandhya Krishnan, Chi chung Lam, Qingda Lu, Marcel Nooijen, Russell M. Pitzer, J. Ramanujam, P. Sadayappan, Alexander Sibiryakov, D. E. Bernholdt, A. Bibireata, D. Cociorva, X. Gao, S. Krishnamoorthy, and S. Krishnan. Synthesis of high-performance parallel programs for a class of ab initio quantum chemistry models. In *Proceedings of the IEEE*, page 2005, 2005.

5 Toby Bloom and Stanley B. Zdonik. Issues in the design of object-oriented database programming languages. In *Proceedings of the Conference on Object-oriented Programming Systems, Languages and Applications*, OOPSLA'87, pages 441–451, 1987.

6 Rastislav Bodík and Barbara Jobstmann. Algorithmic program synthesis: introduction. *STTT*, 15(5-6):397–411, 2013.

7 Bryan Catanzaro, Shoaib Kamil, Yunsup Lee, Krste Asanović, James Demmel, Kurt Keutzer, John Shalf, Ka thy Yelick, and Armando Fox. SEJITS: Getting productivity and performance with selective embedded JIT specialization. In *Workshop on Programming Models for Emerging Architectures (PMEA 2009)*, Raleigh, NC, October 2009.

8 Hassan Chafi, Arvind K. Sujeeth, Kevin J. Brown, HyoukJoong Lee, Anand R. Atreya, and Kunle Olukotun. A domain-specific approach to heterogeneous parallelism. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP'11, New York, NY, USA, 2011. ACM.

9 Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. Optimizing database-backed applications with query synthesis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 3–14, 2013.

10 M. Christen, O. Schenk, and H. Burkhart. Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 676–687, May 2011.

11 Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

12 The django project. `http://www.djangoproject.com`.

13 Sumit Gulwani. Dimensions in program synthesis. In *Proceedings of 10th International Conference on Formal Methods in Computer-Aided Design*, page 1, 2010.

14 Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Synthesis of loop-free programs. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 62–73, 2011.

15 Hibernate. `http://hibernate.org`.

16 F. Irigoin and R. Triolet. Supernode partitioning. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL'88, pages 319–329, New York, NY, USA, 1988. ACM.

17 Java Persistence API. `http://jcp.org/aboutJava/communityprocess/final/jsr317/index.html`.

18 Shoaib Kamil, Cy Chan, Leonid Oliker, John Shalf, and Samuel Williams. An auto-tuning framework for parallel multicore stencil computations. In *IPDPS*, pages 1–12, 2010.

19 David Maier. Representing database programs as objects. In François Bancilhon and Peter Buneman, editors, *Advances in Database Programming Languages*, pages 377–386, New York, NY, USA, 1990. ACM.

20 Saeed Maleki, Yaoqing Gao, Maria J. Garzarán, Tommy Wong, and David A. Padua. An evaluation of vectorizing compilers. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT'11, pages 372–382, Washington, DC, USA, 2011. IEEE Computer Society.

21 Microsoft Entity Framework. `http://msdn.microsoft.com/en-us/data/ef.aspx`.

22 Mohanish Narayan. Polyopt/fortran: A polyhedral optimizer for fortran program. Master's thesis, Ohio State University, 2012.

23 Markus Püschel, José M. F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nicholas Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232– 275, 2005.

24 Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Trans. Graph.*, 31(4):32:1–32:12, July 2012.

25 Tiark Rompf and Martin Odersky. Lightweight modular staging: A pragmatic approach to runtime code generation and compiled dsls. In *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering*, GPCE'10, 2010.

26 Ruby on Rails. `http://rubyonrails.org`.

27 Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 305–316, 2013.

28 Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. Combinatorial sketching for finite programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 404–415, 2006.

29 Yuan Tang, Rezaul Chowdhury, Bradley C. Kuszmaul, Chi keung Luk, and Charles E. Leiserson. The pochoir stencil compiler. In *In SPAA*, 2011.

30 Sid-Ahmed-Ali Touati and Denis Barthou. On the decidability of phase ordering problem in optimizing compilation. In *Proceedings of the 3rd Conference on Computing Frontiers*, pages 147–156, 2006.

31 Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 1–14, Berkeley, CA, USA, 2008. USENIX Association.

32 Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.