Machine Learning-Driven Automatic Program Transformation to Increase Performance in Heterogeneous Architectures

Salvador Tamarit, Guillermo Vigueras, Manuel Carro, and Julio Mariño

Abstract We present a program transformation approach to convert procedural code into functionally equivalent code adapted to a given platform. Our framework is based on the application of guarded transformation rules that capture semantic conditions to ensure the soundness of their application. Our goal is to determine a sequence of rule applications which transform some initial code into final code which optimizes some non-functional properties. The code to be transformed is adorned with semantic annotations, either provided by the user or by external analysis tools. These annotations give information to decide whether applying a transformation rule is or is not sound. In general, there are several rules applicable at several program points and, besides, transformation sequences do not monotonically change the optimization function. Therefore, we face a search problem that grows exponentially with the length of the transformation sequence. In our experience with even small examples, that becomes impractical very quickly. In order to effectively deal with this issue, we have adopted a machine-learning approach using classification trees and reinforcement learning. It learns from successful transformation sequences and produces encodings of strategies which can provide long-term rewards for a given characteristic, avoiding local minima. We have evaluated the proposed technique in a series of benchmarks, adapting standard C code to GPU execution via OpenCL. We have found the automatically produced code to be as efficient as hand-written code generated by an expert human programmer.

S. Tamarit (✉)
Universitat Politècnica de València, València, Spain
e-mail: stamarit@dsic.upv.es

G. Vigueras
IMDEA Software Institute, Madrid, Spain
e-mail: guillermo.vigueras@imdea.org

M. Carro
IMDEA Software Institute, Madrid, Spain
Universidad Politécnica de Madrid, Madrid, Spain
e-mail: manuel.carro@upm.es; manuel.carro@imdea.org

J. Mariño
Universidad Politécnica de Madrid, Madrid, Spain
e-mail: julio.marino@upm.es

C. Niethammer et al. (eds.), *Tools for High Performance Computing 2016*,
DOI 10.1007/978-3-319-56702-0_7

1 Introduction

There is a strong trend in high-performance computing towards the integration of heterogeneous computing elements (vector processors, GPUs, FPGAs, etc.) specially suited for some class of computations. Such platforms are becoming a cost-effective alternative to more traditional supercomputing architectures [4, 12] in terms of performance and energy consumption. This specialization comes at the price of additional hardware and, notably, software complexity. Thus, programming these systems is restricted to a few experts, which hinders its widespread adoption, increases the likelihood of bugs, and limits portability. For these reasons, defining programming models that ease the task of efficiently programming heterogeneous systems has become a topic of great relevance and is the objective of many ongoing efforts.

Many relevant research and industrial projects use scientific code for simulations or numerical solving of differential equations. They often rely on existing algorithms and code that need to be ported to new architectures to exploit their computational strengths to the limit, while at the same time preserving the functional properties of the original code. Unfortunately, and although scientific code commonly follows patterns rooted in its mathematical origin, (legacy) code often does not clearly spell its meaning. In this case, successfully adapting it needs a very careful (and error-prone) transformation process that is hard for humans to do.

Our aim is to obtain a framework for the semantics-preserving transformation of (scientific) C code that improves performance-related metrics on a given destination platform. Despite the broad range of compilation and refactoring tools available, no existing tool fits our goals by being adaptable enough to recognize specific source patterns and generate code better adapted to different architectures. Therefore, we decided to design and implement our own transformation framework. A couple of examples will clarify our motivations and objective.

Figure 1 shows a sequence of program transformation steps to optimize code working on arrays of floats. Some transformation steps can be done by existing optimizing compilers.[1] However, they are usually internally performed at the *intermediate representation* (IR) level, and with few, if any, opportunities for user intervention or tailoring. This falls short to cater for many relevant situations that we want to address:

- In many cases programmers know properties that static analyzers cannot discover. In Fig. 1 a compiler would rely on knowledge of the properties of arithmetic operations (with the caveat in Footnote 1). But if we had calls to functions implementing operations with comparable properties, such as operations

[1] Note, however, that some can not. The standard for floating point arithmetic does not guarantee the preservation of numerical results under the transformation in Step 4 of Fig. 1, and it is therefore not enabled by default in C compilers. However, if this transformation is interesting for some particular domain or application, it can be enabled in our framework by adding the corresponding rule to the ruleset.

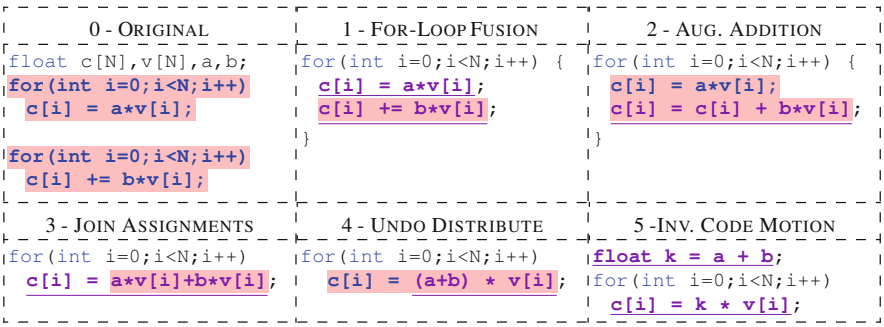

Fig. 1 A sequence of transformations of a piece of C code to compute $\mathbf{c} = a\mathbf{v}+b\mathbf{v}$. `This style` marks code to be modified and `this style` marks code generated from the previous stage

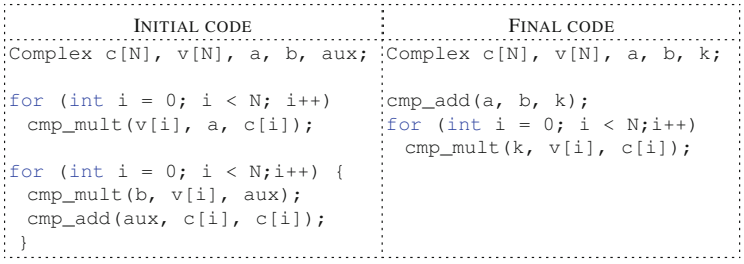

Fig. 2 Transformation enabled by properties similar to those used in Fig. 1

on complex numbers (Fig. 2), the presented transformations would unlikely be performed by a standard compiler.

- Most compilers implement a set of transformations useful for one particular architecture—usually von Neumann-style CPUs. Compiling for a particular architecture needs a specific, ad-hoc compiler that often requires source code to follow some specific guidelines. Our tool can help generate code that complies with these patterns.
- The transformations that generate code amenable to be compiled for specific architectures are often complex, architecture-specific, and domain-specific. Therefore, they are better expressed at a higher level, rather than inside a compiler's architecture, and implemented as extensible plugins.

We aim at generating code that improves some measure of a non-functional characteristic. That needs to select the right rule at every step in the transformation. As part of its modular design, the transformation engine does not have any hard-wired strategy to select which rules have to be applied in each case; instead, it is designed to communicate with external oracles that help in selecting which rules have to be applied. This selection is, however, not without problems. First of all, we require that all the applications are sound—i.e., the (functional) semantics of the code are respected. That needs rules to be applied only when certain conditions

are met. Rules, in our proposal, have guards that express semantic conditions to enable their applications. The code to be transformed is checked to ensure that these conditions are met. As a unifying mechanism, we require that the input code is adorned with *pragmas* expressing properties that cannot be readily derived from the syntactic shape of the code. These pragmas can be inserted by automatic analysis tools or, when they fall short, by the programmer.

Second, when a rule that is part of a sequence that eventually improves some metric is selected and applied, this application may or may not improve that metric. Additionally, at every transformation step several rules can be applied at several points. Therefore, an optimization process may need an exhaustive search in a state space that grows exponentially with the number of steps in the transformation sequence. In our experience, and for relatively small examples, it is typical to have in the order of ten possibilities or more per step and around 50–100 steps in a transformation sequence. That makes exploring the search space unfeasible. In order to deal with that problem we have developed a machine learning-based tool that learns termination conditions and long-range transformation strategies. It is used as an external oracle to select the most promising rule that is part of a transformation chain able to finally improve the code for the target platform. When code deemed good enough for the target architecture is reached, it is handed out to a *translator* that adapts it to the programming model of the target platform.

In the rest of the paper, Sect. 2 reviews previous work in program transformation systems and related approaches using machine learning. Section 3 describes the transformation rule language and properties and the transformation engine. Section 4 discusses the rule selection problem, and Sect. 5 describes a solution based on machine learning. Section 6 presents some preliminary results and, finally, Sect. 7 summarizes the conclusions and proposes future work.

2 Related Work

Stratego-XT [22] is a language-independent transformation tool similar to our proposal, but oriented towards strategies rather than rewriting rules. Rule firing does not depend on semantic conditions that express when applying a rule is sound. This is enough for a language with referential transparency, but not for a procedural one.

CodeBoost [2], built on top of Stratego-XT, performs domain-specific optimizations to C++ code following an approach conceptually similar to ours. User-defined rules express domain-specific optimizations; code annotations are used as preconditions and inserted as postconditions during the rewriting process. However, it is a mostly abandoned project that, additionally, mixes C++, the Stratego-XT language, and their rule language. All of this together makes it to have a steep learning curve. Concept-based frameworks such as Simplicissimus [19] transform C++ based on user-provided algebraic properties. Its rule application strategy can be guided by the cost of the resulting operation, that is defined at the expression level rather than at the statement level and has only a local view of the transformation process. These

issues make its applicability limited and prone to become trapped in local minima (see Sect. 5).

Machine learning techniques have been used for compilation and program transformation [1, 13, 17]. Previous approaches target specific architectures, thereby limiting their applicability and making them unsuitable for heterogeneous platforms. All of them use an abstraction of the input programs, as we do. However, none of the previous works have explored the use of reinforcement learning (RL) methods [9] in the field of program transformation and compilation.

3 Source-to-Source Transformations

The core of the transformation process is a language for defining semantically sound code transformation rules (Sect. 3.1). These rules are fired when some syntactic pattern is found *and* a given semantic property holds. These properties can be either inferred (with the help of an analysis tool) or provided as source code annotations (Sects. 3.2 and 3.3).

3.1 STML Rules

Figure 3 shows a template of a transformation rule. Transformation rules contain a syntactical `pattern` that matches input code and describes the skeleton of the code to `generate`, which will replace the matched code. STML rules (from *Semantic Transformation Meta-Language*) may also specify semantic `conditions` to ensure that their application is sound.[2] As we will see later, these conditions are checked against a combination of static analysis and user-provided annotations in the source code.

Figure 4 shows an example: a rule that applies distributivity "backwards". Pattern components are matched using tagged meta-variables: `e1`, `e2`, and `e3` in the pattern are tagged to specify which kind of component is matched: `cexpr(e1)` states that `e1` must be an expression. These meta-variables are replaced by the matched expression in the generated code. Additional conditions and primitives (Table 1) are used to write sound and expressive rules. In Fig. 4, `pure(cexpr(e1))` means that `e1` is pure, e.g., it does not write to any variable or, in general, it does not perform any state change, including IO. The rule in Fig. 5 performs expression substitution across statements, removing duplicated assignments to variables when possible. In it, `cstmts(`$\mathtt{s}_\mathtt{i}$`)` requires $\mathtt{s}_\mathtt{i}$ to be a sequence of statements. A `cstmt(s)` tag would instead make `s` refer to a single statement.

[2] Properties of the generated code can also be included, but we are not showing them for simplicity.

```
rule_name {
    pattern: {...}
    condition: {...}
    generate: {...}
}
```

Fig. 3 STML rule template

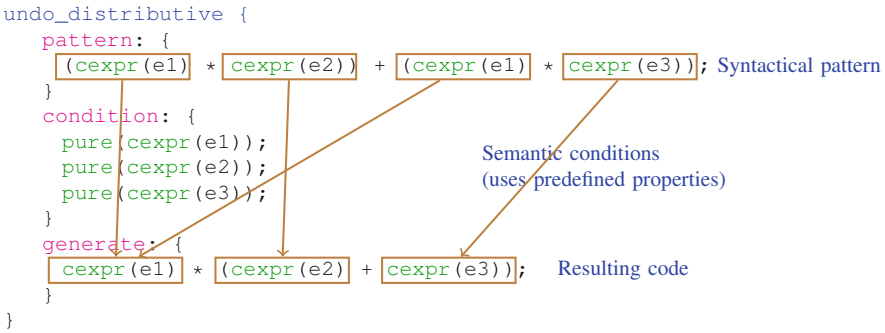

Fig. 4 STML rule: distributive property backwards (steps 3–4 of Fig. 1)

Table 1 presents most of the currently available constructs to write STML rules. In that table, `E` represents an expression, `S` represents a statement and `[S]` represents a sequence of statements. The function `bin_oper(`E_{op}`,`E_l`,`E_r`)` matches or generates a binary operation `(`E_l E_{op} E_r`)` and can be used in the `pattern` and `generate` sections.

The decision of whether to apply or not a given rule depends on two factors: the transformation must preserve the semantics of the transformed code (ensured using the conditions section) and it should eventually improve some efficiency metric. Ensuring the latter is far from trivial, and Sect. 5 will be entirely devoted to our approach to do it effectively. In the next sections we will focus on how to verify that semantic conditions hold before applying a rule.

3.2 *Inferring and Annotating Properties*

Some properties used in the `condition` section can be verified with a local, syntactical check, performed by the transformation engine. However, most interesting conditions need inferring semantic information that requires non-local analysis and we rely on external sources to derive this information. In particular, we are currently using Cetus [5] to this end. Cetus is a compiler framework, written in Java, to implement source-to-source transformations, which we have modified to extract analysis information.

Table 1 Constructs for STML rules

Construct	Description
All sections	
`bin_op(`E_{op}`,`E_1`,`E_2`)`	E_{op} is a binary operation with operands E_1 and E_2
`una_op(`E_{op}`,E)`	E_{op} is a unary operation with operand E
Condition section	
`no_write((S|[S]|E)`$_1$`,(S|[S]|E)`$_2$`)`	`(S|[S]|E)`$_1$ does not write in any location read by `(S|[S]|E)`$_2$.
`no_write_except_arrays ((S|[S]|E)`$_1$`,(S|[S]|E)`$_2$`,E)`	As the previous condition, but not taking arrays accessed using E into account.
`no_write_prev_arrays ((S|[S]|E)`$_1$`, (S|[S]|E)`$_2$`, E)`	No array writes indexed using E in `(S|[S]|E)`$_1$ access previous locations to array reads indexed using E in `(S|[S]|E)`$_2$.
`no_read((S|[S]|E)`$_1$`,(S|[S]|E)`$_2$`)`	`(S|[S]|E)`$_1$ does not read from any location written to by `(S|[S]|E)`$_2$.
`pure((S|[S]|E))`	`(S|[S]|E)` does not write in any location.
`writes((S|[S]|E))`	Locations written by `(S|[S]|E)`.
`distributes_over(`E_1`,`E_2`)`	Operation E_1 distributes over operation E_2.
`occurs_in(E,(S|[S]|E))`	Expression E occurs in `(S|[S]|E)`.
`fresh_var(E)`	E should be a new variable.
`is_identity(E)`	E is the identity.
`is_assignment(E)`	E is an assignment.
`is_subseteq(`E_1`,`E_2`)`	$E_1 \subseteq E_2$
Generate section	
`subs((S|[S]|E),`E_f`,`E_t`)`	Replace each occurrence of E_f in `(S|[S]|E)` for E_t.
`if_then:{`E_{cond}`; (S|[S]|E);}`	If E_{cond} is true, then generate `(S|[S]|E)`.
`if_then_else:{`E_{cond}`; (S|[S]|E)`$_t$`;(S|[S]|E)`$_e$`;}`	If E_{cond} is true, then generate `(S|[S]|E)`$_t$ else generate `(S|[S]|E)`$_e$.
`gen_list:{[(S|[S]|E)];}`	Each element in `[(S|[S]|E)]` produces a different rule consequent.

Instead of devising an internal API to communicate results, all analysis information is passed on to the rewriting engine by annotating the source code with *#pragma*s. A pragma captures properties belonging to the code block immediately following it and the properties range from expression pureness to read/write dependencies in arrays. Figure 6 shows four pieces of code that read and write on arrays with an offset w.r.t. the loop index as expressed by the annotations. For example, Fig. 6b writes in `c[]` in positions `i+0` and `i-1`, with `i` being the loop index. This is expressed with the set `{-1, 0}`. The core syntax of STML annotations is shown for reference in Listing 1, and Table 2 gives a summary overview of higher-level annotations. A more thorough explanation of their semantics is to be found in [20].

It is often the case that automatic analyzers cannot infer all the information necessary to decide the soundness of the application of some rules. In that case, we rely on the programmer to annotate the code by hand using pragmas. That is

```
join_assignments {
   pattern: {
      cstmts(s1);
      cexpr(v) = cexpr(e1);
      cstmts(s2);
      cexpr(v) = cexpr(e2);
      cstmts(s3);
   }
   condition: {
      no_write(cstmts(s2), {cexpr(v), cexpr(e1)});
      no_read(cstmts(s2), {cexpr(v)});
      pure(cexpr(e1));
      pure(cexpr(v));
   }
   generate: {
      cstmts(s1);
      cstmts(s2);
      cexpr(v) = subs(cexpr(e2), cexpr(v), cexpr(e1));
      cstmts(s3);
   }
}
```

Fig. 5 STML rule: assignment propagation (steps 2–3 of Fig. 1)

```
#pragma stml writes c in {0}
for (i = 0; i < N; i++)
   c[i] = i*2;
```

(a)

```
#pragma stml writes c in {-1,0}
for (i = 1; i < N; i++){
   c[i-1] = i;
   c[i]  = c[i-1] * 2;}
```

(b)

```
#pragma stml reads c in {0}
for (i = 0; i < N; i++)
   a += c[i];
```

(c)

```
#pragma stml reads c in {-1,0,+1}
for (i = 1; i < N - 1; i++)
   a += c[i-1]+c[i+1]-2*c[i];
```

(d)

Fig. 6 Code with STML annotations

one reason to use them as interface to communicate information to the rewriting engine: information becomes available in a uniform format regardless of its origin. If the annotations automatically inferred by external tools contradict those provided by the user, the properties provided by the user are preferred to those deduced from external tools, but a warning is issued.

3.3 High-Level Annotations

STML annotations can capture very detailed information regarding code properties and programmers can fill in the gaps when automatic analysis is not enough. How-

Listing 1 BNF grammar core for STML

```
<code_prop_list> ::= "#pragma stml" <code_prop> |
                     "#pragma stml" <code_prop> <code_prop_list>
<code_prop>  ::= <loop_prop> | <exp_prop> <exp> |
                 [<op>] <op_prop> <op> |
                  "write("<exp>") =" <location_list> |
                  "same_length" <exp> <exp> | "output("<exp>")" |
                  <mem_access> <exp> ["in" <offset_list>]
<loop_prop>  ::= "iteration_independent" |
                  "iteration_space" <parameter> <parameter>
<exp_prop>   ::= "appears" | "pure" | "is_identity"
<op_prop>    ::= "commutative" | "associative" |
                  "distributes_over"
<mem_access> ::= "writes" | "reads" | "rw"
```

Table 2 Intuitive meaning of STML annotations

`write(exp) = loc`	expression `exp` writes in location `loc`
`writes exp`	the block below write in a location identified by `exp`
`writes exp in offsets`	the block below write in the set of locations identified by array `exp` with `offsets` w.r.t. a loop index
`iteration_space exp1 ex2`	the index of the annotated loop ranges from exp1 to exp2
`iteration_independent`	loop iterations are independent from each other
`same_length a1 a2`	arrays `a1` and `a2` have the same length
`input exp`	`exp` is to be seen as an `input` of the following code block
`output exp`	`exp` is to be seen as an `output` of the following code block
`appears exp`	`exp` appears in the block below
`pure exp`	`exp` does not update any variable
`is_identity exp`	`exp` is an identity element
`commutative op`	`op` is commutative
`associative op`	`op` is associative
`op1 distributes_over op2`	self-explanatory

ever, the type of information necessary is not what a programmer has naturally in mind, and the amount of annotations necessary may exceed what can be considered as an acceptable effort. Therefore, we also accept a second level of annotations that were devised as part of the POLCA project.[3] They have, intuitively, a more

[3] http://www.polca-project.eu/.

```
float c[N], v[N], a, b;

#pragma polca map BODY1 v c
for(int i=0;i<N;i++)
#pragma polca def BODY1
#pragma polca input v[i]
#pragma polca output c[i]
  c[i] = a*v[i];
```

```
#pragma polca zipWith BODY2 v c c
for(int i=0;i<N;i++)
#pragma polca def BODY2
#pragma polca input v[i]
#pragma polca input c[i]
#pragma polca output c[i]
  c[i] += b*v[i];
```

Fig. 7 Annotations for the code in Fig. 1

```
float c[N], v[N], a, b;

#pragma polca map BODY1 v c
#pragma stml reads v in {0}
#pragma stml writes c in {0}
#pragma stml same_length v c
#pragma stml pure BODY1
#pragma stml iteration_space 0 length(v)
#pragma stml iteration_independent
for(int i = 0; i < N; i++)
#pragma polca def BODY1
#pragma polca input v[i]
#pragma polca output c[i]
  c[i] = a*v[i];
```

```
#pragma polca zipWith BODY2 v c c
#pragma stml reads v in {0}
#pragma stml reads c in {0}
#pragma stml writes c in {0}
#pragma stml same_length v c
#pragma stml pure BODY2
#pragma stml iteration_space 0 length(v)
#pragma stml iteration_independent
for(int i = 0; i < N; i++)
#pragma polca def BODY2
#pragma polca input v[i]
#pragma polca input c[i]
#pragma polca output c[i]
  c[i] += b*v[i];
```

Fig. 8 Translation of high-level annotations in Fig. 7 into STML

algorithmic appearance (they are actually inspired by functional programming [10]) and capture simultaneously algorithm skeletons and low-level properties.

For instance, `for` loops performing a mapping between an input and an output array can be annotated with a `map` pragma (see one example in Fig. 7, left). The scheme for a `map` annotation is

```
#pragma polca map Func Input Output
```

where `Func` stands for the name of a block of code and `Input` and `Output` are names of array variables. The `map` annotation in Fig. 7 indicates that the loop traverses the input array `v` and applies the function computed by `BODY1` element-wise to `v` giving as result the (output) array `c`. Besides this algorithmic view, the annotation also implies several properties of the code: (a) `BODY1` behaves as if it were side effect-free (it may read and write from/to other variables not declared as parameters, but it should behave as if these variables did not implement a state for `BODY1`), (b) `v` and `c` are arrays of the same size, (c) every element `c[i]` is computed by applying `BODY1` to `v[i]`, (d) the applications of `BODY1` do not assume any particular order: they can go from `v[0]` upwards to `v[N-1]`, in the opposite direction, or in any other order.

These properties have a counterpart in STML and are the kind of conditions that the transformation engine checks: it reads the high-level pragmas and transforms them into STML for internal use. As an example, Fig. 8 shows the translation of the code in Fig. 7 into STML. The difference between them supports our claim that high-level annotations make annotating the program easier and can convey a large amount of relevant information.

3.4 *Implementation Notes*

The transformation engine is subdivided into two subcomponents, illustrated in Fig. 9. The rule-driven code transformation stage proper changes the structure of the code until it has the patterns appropriate for the destination architecture and produces what we call *ready code*. Note that this transformation stage can additionally be used to other purposes, such as sophisticated code refactoring. A second code translation stage converts this code into the input language for a compiler for the destination architecture. This last translation stage is in many cases straightforward as it only introduces the "idioms" necessary for the architecture (e.g., for OpenMP), performs a syntactical translation (e.g., for OpenCL) or mixes both (e.g. for ROCCC [8]), but some targets (e.g., MaxJ [15]) are admittedly more involved. The particular target architecture is specified with an annotation, which is also used to decide what transformations should be applied.

The transformation phase is a key part of the tool. In order to be able to experiment and prototype as easily as possible, (including the STML definition, code generation, and the search/rule selection procedures), we needed a flexible and expressive implementation platform. We considered using the infrastructure provided by existing open source C compilers. Among these, the CLang / LLVM

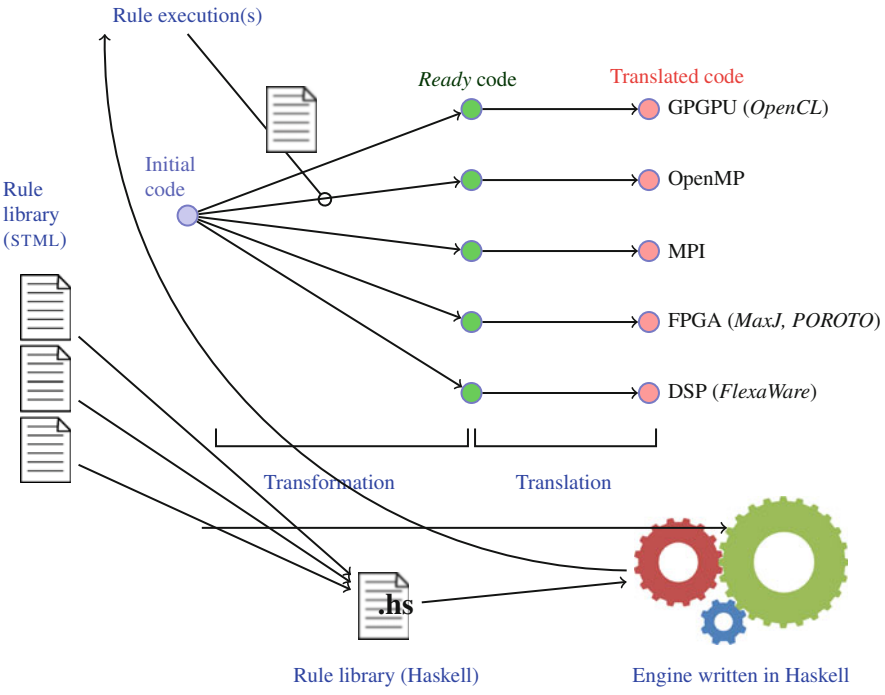

Fig. 9 Architecture of the transformation tool

libraries/APIs have probably the best design. However, since their goal is compilation rather than source-to-source transformation, we found the available interface neither easy to use nor effective in many situations. Moreover, existing documentation warns about its instability. Additionally, code transformation routines had to be coded in C++, which made them verbose and full of low-level details, contrary to the flexibility we needed. Compiling rules to C++ was an option, but the gap between the rules and the API was quite large, pointing to a cumbersome translation stage that would require considerable maintenance as the rule language evolved. Moreover, the whole CLang project needed recompilation after every rule update. That would have made project development and testing very slow, and adding user-defined rules complicated.

We decided therefore to use a declarative language and we implemented the transformation engine in Haskell. Parsing the input code and the rules is done by means of the `Language.C` library [6] that returns the AST as a data structure that is easy to manipulate. In particular, we used the Haskell facilities to deal with generic data structures through the *Scrap Your Boilerplate* (SYB) library [11]. This allows us to easily extract information from the AST or modify it with a generic traversal of the whole structure.

The rules are written in a subset of C and are parsed using `Language.C`. They are compiled into Haskell code (contained in the file `Rules.hs`—see Fig. 9) that performs the traversal and (when applicable) the transformation of the AST. This module is loaded with the rest of the tool, therefore avoiding the extra overhead of interpreting the rules.

Rule compilation divides rules into two classes: those that operate on expressions and those that can, in addition, manipulate sequences of statements. In the latter case, sequences of statements of unknown length need to be considered: for example, `s1`, `s2`, and `s3` in Fig. 5. In general, the rule has to try several possibilities to determine if there is a match that meets the rule conditions. Haskell code that explicitly performs an AST traversal has to be generated. Expressions, on the other hand, are syntactically bound and the translation of the rule is much easier.

4 Rule Selection

In most cases, several (often many) rules can be safely applied at multiple code points in every step of the rewriting process. Deciding which rule has to be fired should be ultimately decided based on whether it eventually increases performance. We currently provide two rule selection mechanisms: a human interface and a API to communicate with external tools.

The human interface allows making interactive transformations possible. The user is presented with the rules that can be applied at some point and, after selecting a rule, the code before and after applying it. Auxiliary programs, such as *Meld*,[4] can

[4] http://meldmerge.org/.

$$AppRules(Code) \rightarrow \{(Rule_i, Pos_i)\}$$
$$Trans(CodeI, Rule, Pos) \rightarrow CodeO$$

Fig. 10 Functions provided by the transformation tool

$$SelectRule(\{(Code_i, \{Rule_j\})\}) \rightarrow (CodeO, RuleO)$$
$$IsFinal(Code) \rightarrow Boolean$$

Fig. 11 Functions provided by the oracle

Header

$NewCode(CodeI, \{Rule_i\}) \rightarrow (CodeO, RuleO)$

Definition

$$NewCode(c, rls) = SelectRule(\{(c', \{r' \mid (r', \_) \in AppRules(c')\} \mid c' \in \{Trans(c, r, p) \mid (r, p) \in AppRules(c), r \in rls\}\})$$

Complete derivation

$$NewCode(c_0, AllRules) \rightarrow^* (c_n, r_n) \text{ when } IsFinal(c_n)$$
$$\forall i, 0 < i < n \cdot (c_i, r_i) = NewCode(c_{i-1}, \{r_{i-1}\}) \text{ when } \neg IsFinal(c_i)$$

Fig. 12 Interaction between the transformation and the oracle interface

be used to highlight the differences. This is useful to refine/debug rules or to perform general-purpose refactoring. However, in our experience, manual rule selection is not scalable when working in adapting code to a given platforms, and using it is not feasible even for medium-sized programs. Therefore, mechanizing this process as much as possible is a must and we designed a general interface to connect external components. Regardless of how such an external component works, from the point of view of the transformation engine it is an *oracle* that, given some code and a set of applicable rules, returns which rule should be applied.

The interface of the transformation tool (Fig. 10) is composed by functions *AppRules* and *Trans*. The former determines the possible transformations applicable to a given input code *Code* and returns a set of tuples containing each a rule name and the position (e.g., the identifier of an AST node) where the rule can be applied. Function *Trans* applies rule *Rule* to code *CodeI* at position *Pos* and returns the transformed code *CodeO*.

The API from the external tool (Fig. 11) includes operations to decide which rule has to be applied and whether the search should stop. Function *SelectRule* receives a set of safe possibilities, each of them composed of a code fragment and a set of rules that can be applied to it, and returns one of the input code fragments and the rule that should be applied to it. Function *IsFinal* decides whether a given code can be considered ready for translation or not.

In Fig. 12, function *NewCode* sketches how the interaction between the transformation engine and the external oracle takes place. In a nutshell, *NewCode* is invoked with code to be transformed and generates transformed code which is, in turn, iteratively passed to *NewCode* until a termination condition is fulfilled (i.e., *IsFinal* evaluates to true), and the generated code is then final. In more detail, *NewCode*

receives input code in the parameter *CodeI* and a set of (candidate) transformation rules $\{Rule_i\}$ and returns: (a) one piece of transformed code (*CodeO*) and (b) one rule (*RuleO*). When the transformation is not finished, *NewCode* is called again with the transformed code *CodeO* and with the singleton set of rules $\{RuleO\}$. Therefore, when *NewCode* applies a transformation, the oracle decides which rule should be applied next to the just-generated transformed code.

This approach makes it unnecessary for the external oracle to consider code positions where a transformation can be applied, since that choice is implicit in the selection of a candidate code between all possible code versions obtained using a single input rule. Furthermore, by selecting the next rule to be applied, it takes the control of the next step of the transformation. The key here is the function *SelectRule*: given inputs $Code_i$ and $\{Rule_j\}$, *SelectRule* selects a resulting code between all the codes that can be generated from $Code_i$ using $Rule_j$. The size of the set received by function *SelectRule* corresponds to the total number of positions where $Rule_j$ can be applied. In this way, *SelectRule* is implicitly selecting a position.

5 Controlling the Transformation Process with Machine Learning

Several outstanding problems are faced by the rewriting engine. On the one hand, the space of transformation sequences leading to different code versions is very large (actually infinite) and the only guide is a non-monotonic fitness function (e.g., performance) very costly and cumbersome to evaluate. On the other hand, deciding when a sequence finishes is difficult to check: the final state is reached when the most efficient possible code has been generated.

Selecting at each step a rule that improves more some metric is not sound: code performance along good transformation sequences evolves non-monotonically.[5] This non-monotonicity can make the search be trapped in local minima. In addition, and as another face of non-monotonicity, the performance of ready code is not correlated with that of the code translated for the final architecture, so ready code cannot be used to make reliable predictions of final performance. Exploring a bounded neighborhood is not a satisfactory solution, either, since a large boundary would have efficiency problems and a small boundary would not avoid the local minima problem.

Therefore, we need a mechanism that can make local decisions taking into account global strategies—i.e., a procedure able to select a rule under the knowledge that it is part of a larger sequence that will eventually improve code performance for a given platform. Our approach uses classification trees to decide when to finish a

[5]Not only in theory: in our experience, it is often necessary to apply transformations that temporarily reduce performance because they enable further transformations.

transformation sequence and reinforcement learning to select which transformation rule has to be applied at every moment. We will describe our approaches in the next sections.

5.1 Mapping Code to Abstractions

Machine learning operates on descriptions of the problem domain (C code, in our case). They have to be able to capture the changes performed by the transformation rules at the AST level and represent code patterns that match the syntactic/semantic restrictions of target compiler/programming models in order to decide when a transformation sequence can finish. For these reasons, the abstraction includes quantitative descriptions involving features like AST patterns, control flow, data layout, data dependencies, etc. The current abstraction consists of a vector of features shown in Table 3 and a short explanation of some of them follows:

- **Number of auxiliary array variables:** number of auxiliary variables used to index an array. For Listing 2 its value would be "one".
- **All loops have static limits:** it is false iff some `for` loop in the analyzed code has a non-static iteration limit. It would be `false` for Listing 3, since `clean` or `update` could change the data structure.
- **Scheduled loop:** two nested loops iterate over an array "split in fragments". This is deduced from the annotation in Listing 4.
- **Shifted writes in array:** number of loops where some (but not all) writes to arrays have a positive offset w.r.t. the iteration variable. It would have a value of "one" in Listing 5.

Table 3 Features currently used in the learning process

Description	Type
Maximum depth among nested `for` loops	$\mathbb{N}$
Number of function calls present in the analyzed code	$\mathbb{N}$
Number of array accesses with positive offset in bodies of `for` loops	$\mathbb{N}$
Are there loops with non-structured flow?	$\mathbb{B}$
Is any global variable written on?	$\mathbb{B}$
Number of `if` statements	$\mathbb{N}$
Has any `for` loop a non-static iteration limit?	$\mathbb{B}$
Number of `for` loops without dependencies across iterations	$\mathbb{N}$
Whether two nested loops iterate over an array split in fragments	$\mathbb{B}$
Number of variables used inside a loop and unmodified inside it	$\mathbb{N}$
Number of variables modified within a loop	$\mathbb{N}$
Number of arrays with two or more dimensions	$\mathbb{N}$
How many auxiliary variables are used to index arrays	$\mathbb{N}$
Total number of `for` loops	$\mathbb{N}$
Number of `for` with iteration step different from 1	$\mathbb{N}$

Listing 2 Aux. variable array index

```
aux = 0;
for(j=0; j<N; j++) {
  w[j] = v[aux];
  aux++;
}
```

Listing 3 Static loop limits

```
for(j=0;j<N;j++) {
  for(i=0;i<size(v);i++)
    update(v,i);
  clean(v);
}
```

Listing 4 Loop with schedule pattern

```
#pragma stml loop_schedule
for(j=0; j<M; j++) {
  w[j] = 0;
  for(i=0;i<N;i++)
    w[j] += v[j*N+i];
}
```

Listing 5 Array writes shifted

```
for(i=1;i<N;i+=2) {
  v[i] = v[i-1];
  v[i+1] = v[i-1]*i;
}
```

The code abstraction is generated through an analysis tool that parses the AST to extract the abstraction features, thereby implementing a function

$$A : \mathit{Code} \rightarrow \mathit{Abstraction}$$

that maps codes to abstractions. In order to simplify communication with the rest of the machine learning component, that uses Python libraries, the code abstraction extraction is also implemented in Python using the *pycparser*[6] module. It extracts features both by analyzing the code and by parsing code annotations. The current set of features were enough to obtain the results for our current set of use cases (Sect. 6).

5.2 Deciding Termination with Classification Trees

Classification is the problem of identifying the category to which a new observation belongs among a set of pre-defined categories. Classification is done by training using a set of observations for which it is known to which category they belong [14]. Among the existing approaches, we have evaluated classification trees since it can perform feature selection without complex data preparation.

A classification tree organizes examples according to a set of input features belonging to finite discrete domains. One of the features is the *target variable* and the classification tree aims at inferring its value from the values of the rest of the features. Each element of the domain of the target variable is called a class.

In a classification tree each non-leaf node is labeled with an input feature and each leaf node is labeled with a class or a probability distribution over the classes. A classification tree can be built by splitting the source data set into subsets based

[6] https://github.com/eliben/pycparser.

on values of input features and recursively repeating the process on each derived subset. The recursion finishes when the subset of data in a node has the same value for the target variable or when splitting no longer improves the predictions. The source data typically comes in records of the form

$$([x_1, x_2, x_3, \ldots, x_k], Y)$$

Y is the target variable that the classification tree generalizes in order to be able to classify new observations. The elements x_i are the input features used for the classification, drawn from those in Table 3. The target variable determines to what platform(s) the code can be translated. Since a given code (and its associated abstraction) might be suited for more than one platform, for n platforms we have $2^n - 1$ classes in the target variable. In our current setup, since we currently support FPGAs, GPUs, Shared-Memory CPUs, and Distributed-Memory CPUs, we have 15 elements in the domain of Y.

The classes obtained for the target variable define the final states for the reinforcement learning algorithm described next. The classification-based learning described in this section has been implemented using the Python library *Scikit-learn* [16]. This library implements several machine learning algorithms, provides good support and ample documentation, and is widely used in the scientific community.

5.3 Reinforcement Learning

Reinforcement learning [14] is an area of machine learning whose aim is to decide how software agents ought to act to maximize some notion of cumulative reward. A reinforcement learning agent interacts with its environment in discrete time steps. At time t, the agent receives an observation o_t that typically includes a reward r_t. It then chooses an action a_t that is sent to the environment which changes from state s_t to state s_{t+1} providing the reward r_t associated with the transition (s_t, a_t, s_{t+1}). The goal of a reinforcement learning agent is to collect as much reward as possible.

RL seems well suited to represent the process of a programmer or a compiler: iteratively improving an initial program in discrete steps. Actions correspond to code changes (caused in our case by the application of transformation rules) and states correspond to code versions. Code can in principle be evaluated after every change according to properties such as execution time, memory consumption speedup factor, etc. The result of these evaluations can be translated into rewards and penalties that feed the learning procedure.

The final result of the learning process of the agent is a *state-action* table Q (Fig. 15) that contains, for each combination (s, a) of states and actions, the expected profit to be obtained from applying action a to state s. This table is initially filled in with a default value and is iteratively updated following a learning process that we briefly describe below.

Reinforcement learning uses a set of predetermined transformation sequences that are assumed to be models to learn from. Each sequence S is composed of a set of states $S = s_0, s_1, \ldots, s_l$ and the actions that transform one state into the next one. The final state of each transformation sequence has a different reward related to the performance of corresponding code. The training phase of reinforcement learning consists of an iterative, stochastic process where a state s from the training sequences is randomly selected and a *learning episode* is started by selecting the action a with the highest value in Q for that s. The learning process moves to a new state s' according to the transition (s, a, s') and the process is repeated from state s' until a final state is reached or a given number of steps is performed. When the episode terminates, the values in Q corresponding to the states and actions of the visited sequence are updated according to the formula in Fig. 13, where $Q_{init}(s_t, a_t)$ is the initial value of Q for state s_t and action a_t (resp. $Q(s_t, a_t)$). Note that s_t (resp. a_t) is the t-th state in the temporal ordering of states in the sequence used to learn.

The final states in Fig. 13 are defined based on the classification described in Sect. 5.2. Two parameters appear in Fig. 13: the learning rate $\alpha, 0 < \alpha \leq 1$, and the discount factor $\gamma, 0 < \gamma \leq 1$. The learning rate determines to what extent the newly acquired information will override the old information. A factor of 0 will make the agent not to learn anything while a factor of 1 would make the agent consider only the most recent information. The discount factor determines the importance of future rewards, and so it implements *delayed rewards*. A factor of 0 will make the agent opportunistic by considering only current rewards and a factor close to 1 will make it strive for long-term rewards. If the discount factor reaches or exceeds 1, the learning process may diverge [14].

Code abstractions and transformation rules are mapped to states and actions, respectively, to index the *state-action* table, using functions *SM* and *AM* (Fig. 14). Using the mapping of abstractions and rules to states and actions, the *state-action* table can also be modeled as a function Q ranging over code and rules (Fig. 14). The rule selection strategy of the transformation toolchain can then be modeled with function *RS* that takes as input a code c and selects the transformation rule r associated to action $AM(r)$ that maximizes the value provided by Q for the state $SM(A(c))$ associated to input code c.

$$Q(s_t, a_t) = \begin{cases} Q(s_t, a_t) + \alpha \cdot (r_{t+1} + \gamma \cdot Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)) & \text{if } s_t \text{ not final} \\ Q_{init}(s_t, a_t) & \text{otherwise} \end{cases}$$

Fig. 13 Update of reinforcement learning matrix

$SM: Abstraction \rightarrow State$	$Q: State \times Action \rightarrow \mathbb{R}$
$AM: Rule \rightarrow Action$	$RS: Code \rightarrow Rule$
$RS(c) = \arg\max_{ru \in Rule} Q(SM(A(c)), AM(ru))$	

Fig. 14 RL function definitions

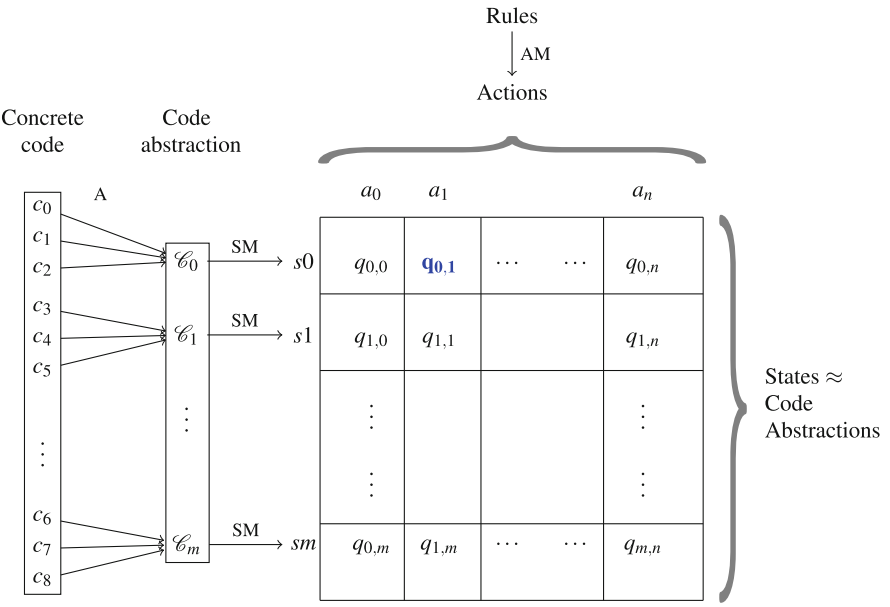

Fig. 15 State-Action table for code, code abstraction, and rules

The operator arg max in function *RS* returns, by definition, a set that can be empty or non-singleton. However, in our problem, parameters α and γ, as well as the reward values r_{t+1}, can be tuned to ensure that a single rule is returned, thus avoiding a non-deterministic *RS* function. The workflow is then as follows (Fig. 15): for a concrete code c_k we find its abstraction $\mathscr{C}_i = A(c_k)$. Let us assume $i = 0$. From the row i we obtain the column j with the highest value $q_{i,j}$ in matrix Q (in our example, $q_{0,1}$, in blue and boldface). Column j corresponds to rule r_j, which is expected to give the next step in the most promising sequence when applied to a code state whose abstraction is $\mathscr{C}_i$ (in our case it would be r_1). Rule r_j would be applied to c_k to give c_l. If c_l corresponds to a final state, the procedure finishes. Otherwise, we repeat the procedure taking c_l as input and finding again a rule to transform c_l.

We have implemented the reinforcement learning component using the Python library *PyBrain* [18]. This library adopts a modular structure separating in classes the different concepts present in reinforcement learning, such as the environment, the observations and rewards, the actions, etc. This modularity allowed us to extend the different classes and ease their adaptation to our problem. The *PyBrain* library also provides flexibility to configure the different parameters of the reinforcement learning algorithm.

5.4 A Simple Example

We will use a 2D convolution kernel (Listing 6) to show the process of learning a state-action table from a transformation sequence. This kernel can already be executed in parallel by adding OpenMP pragmas. However, adapting it to target platforms like GPUs or FPGAs requires a different set of transformations. For example, by joining the two outer loops, to obtain a linear iteration space or transforming the data layout of 2D arrays into 1D arrays, we obtain a sequential code easier to map onto the two platforms mentioned before.

We will use five states (Listings 6–10) and two transformation rules to showcase how these transformations can be executed. The first rule (R_0) transforms a non-1D array into a 1D array and the second rule (R_1) collapses two nested `for` loops into a single loop. Color codes are as in Fig. 1.

Listing 6 Initial code

```
//[3,0,0,0,0,0,1,0,0,1,1,3,2,4,0]
for (r = 0; r < N - K + 1; r++)
  for (c = 0; c < N - K + 1; c++) {
    sum = 0;
    for (i = 0; i < K; i++)
      for (j = 0; j < K; j++)
        sum += img_in[r+i][c+j] * kernel[i][j];
    img_out[r+dead_rows][c+dead_cols] = (sum /
        normal_factor);
  }
```

Listing 7 Transformation step 1

```
// [3,0,0,0,0,0,1,0,0,1,1,2,2,4,0]
for (r = 0; r < N - K + 1; r++)
  for (c = 0; c < N - K + 1; c++) {
    sum = 0;
    for (i = 0; i < K; i++)
      for (j = 0; j < K; j++)
        sum +=
          img_in[(r+i)*(N-K+1)+(c+j)]
              * kernel[i][j];
    img_out[r+dead_rows][c+dead_cols]
        = (sum / normal_factor);
  }
```

Listing 8 Transformation step 2

```
// [3,0,0,0,0,0,1,0,0,1,1,1,2,4,0]
for (r = 0; r < N - K + 1; r++)
  for (c = 0; c < N - K + 1; c++) {
    sum = 0;
    for (i = 0; i < K; i++)
      for (j = 0; j < K; j++)
        sum +=
          img_in[(r+i)*(N-K+1)+(c+j)]
              * kernel[i*K+j];
    img_out[r+dead_rows][c+dead_cols]
        = (sum / normal_factor);
  }
```

Listing 9 Transformation step 3

```
// [3,0,0,0,0,0,1,0,0,1,1,0,2,4,0]
for(r = 0; r < N - K + 1; r++)
  for(c = 0; c < N - K + 1; c++) {
    sum = 0;
    for (i = 0; i < K; i++)
      for (j = 0; j < K; j++)
        sum +=
          img_in[(r+i)*(N-K+1)+(c+j)]
              * kernel[i*K+j];
    img_out[(r+dead_rows)*(N-K+1) +
        (c+dead_cols)] =
      (sum / normal_factor);
  }
```

Listing 10 Transformation step 4

```
// [2,0,0,0,0,0,0,1,1,0,1,1,0,2,3,0]
for(z=0; z<(N-K+1)*(N-K+1); z++) {
  int r = (z / (N - K + 1));
  int c = (z % (N - K + 1));
  sum = 0;
  for (i = 0; i < K; i++)
    for (j = 0; j < K; j++)
      sum +=
        img_in[(r+i)*(N-K+1)+(c+j)]
            * kernel[i*K+j];
  img_out[(r+dead_rows)*(N-K+1) +
        (c+dead_cols)] =
    (sum / normal_factor);
}
```

	$AM(R_0)$	$AM(R_1)$	$RS(C_i)$
$SM(A(C_0))$	17.03718317	16.21544456	R_0
$SM(A(C_1))$	17.25327145	16.80486418	R_0
$SM(A(C_2))$	17.51541052	16.7189079	R_0
$SM(A(C_3))$	16.72942327	17.78007298	R_1
$SM(A(C_4))$	1.	1.	-

Fig. 16 Values learned for Q table

Every listing shows, at the beginning, the feature vector marking the feature component that changed w.r.t. the previous state. In Listings 7 to 9, rule R_0 is applied to Listing 6 to transform 2-D arrays `img_in`, `kernel`, and `img_out` (in this order) into 1-D arrays. Listing 10 shows the result of applying rule R_1, which collapses the two outermost loops into one `for` loop keeping an iteration space with the same number of iterations.

Figure 16 shows a table with the final *state-action* table Q for the transformation sequence described before, obtained as the result of the learning process described before. The table has a column for each applied rule and a row for each state corresponding to the code versions in the learning sequence. The values in blue mark the learned sequence (the highest value in each row), composed of three applications of rule R_0 and one application of rule R_1. These values decrease from the state $SM(A(C_3))$ down to the initial state $SM(A(C_0))$. This decay behavior is caused by the discount factor γ. The values in Q for the final states are not updated by the recursive expression in Fig. 13 and therefore the state $SM(A(C_4))$ keeps its initial value.

We have seen the transformations applied to C code. However, since machine learning methods work on program abstractions, the approach is very generic and suitable for other imperative languages (e.g., FORTRAN). Applying our approach to other languages would require changes to accommodate for language-specific syntactic patterns. Nevertheless, the abstraction features described in Sect. 5.1 capture common aspects like control flow, data layout, data dependencies, etc. and can therefore be applied to other imperative languages with little effort.

6 Results

We will evaluate our proposal on a set of image processing-related benchmarks. We will first show the non-monotonic behavior of non-functional properties for good transformation sequences and, second, we will evaluate the effectiveness of reinforcement learning to learn from these non-monotonic sequences and apply the learned knowledge.

We will illustrate the non-monotonic behavior of performance characteristics with four transformation sequences applied to code for the discrete cosine transform. These four sequences finish by producing C code than can be straightforwardly translated into OpenCL. We have measured the average execution time of 30 runs for each intermediate state of each sequence and represented them in Fig. 17, where the non-monotonic behavior is clear.

Next, we translated into OpenCL the code corresponding to the final states in Fig. 17 and we compared its performance against the original C code (Fig. 18). The fastest OpenCL version corresponds to sequence 4; however, Fig. 17 reveals that the *ready code* for sequence 4 was actually the *second slowest* one on a CPU. In fact, comparing Figs. 17 and 18, there does not seem to be any clear correlation between the execution time of the ready code and the performance of the corresponding OpenCL version. We hypothesize that the same would happen to other non-functional properties. Based on these results we conclude that an effective method to automatically generate high-performance code must discover (and learn) uncorrelated relations between code behavior on CPUs and on target platforms. That is one of the reasons to base our approach on reinforcement learning, since it is driven by final performance measurements rather than on intermediate values.

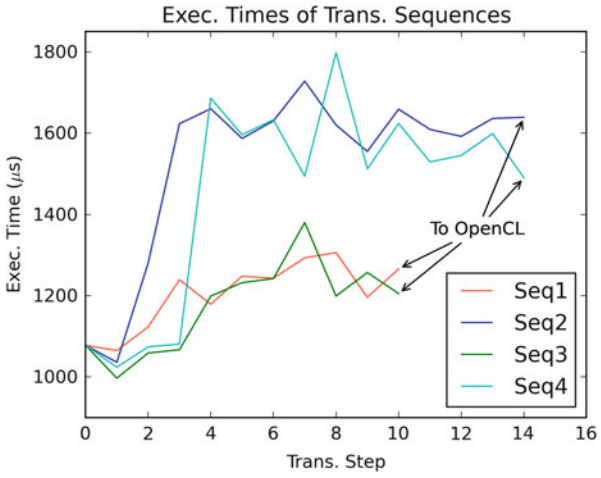

Fig. 17 Execution times for transformation sequences (on a CPU)

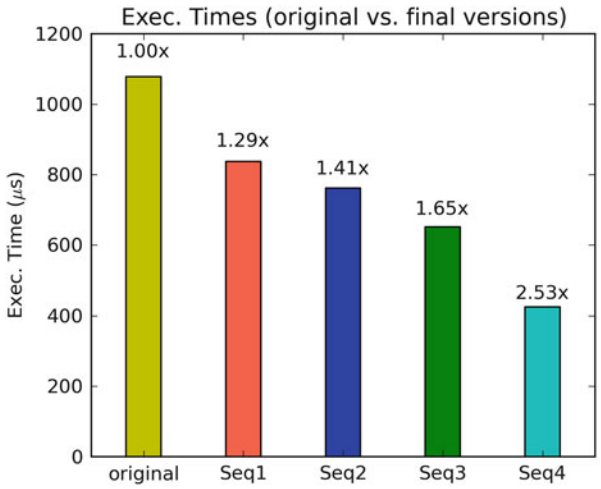

Fig. 18 Execution times for OpenCL versions (on a GPU)

To evaluate reinforcement learning as a technique to learn and guide our program transformation component, we have selected a training set of four benchmarks targeting OpenCL. The training set contains the image compression program (*compress*), an image filter that separates an RGB image into different images for each color channel (*rgbFilter*), an image edge detection routine using a Sobel filter (*edgeDetect*), and code for image segmentation given a threshold value (*threshold*).

Once the training set is defined, the reinforcement learning process requires tuning the learning rate (α) and the discount factor (γ). We experimentally adjusted them to values leading to transformation sequences providing the fastest OpenCL versions (with $\alpha = 0.5$ and $\gamma = 0.6$). The reward values were chosen to reinforce sequences leading to code with better performance. In our case we gave them a reward 100 times bigger than that of the other sequences.

After training, three different applications were used as prediction set; these were mechanically transformed according to the previously learned sequences and finally translated into OpenCL. The prediction set shares code patterns with the training set; this is aligned with the idea that transformation rules can be tailored to the application domain. Independently, OpenCL versions of the prediction set were manually written to compare automatically- and manually-generated code. Figure 19 shows the results: the automatically generated code provides speedup factors comparable to the manually coded versions. Although this preliminary evaluation is based on a small sample, it shows that our approach is promising.

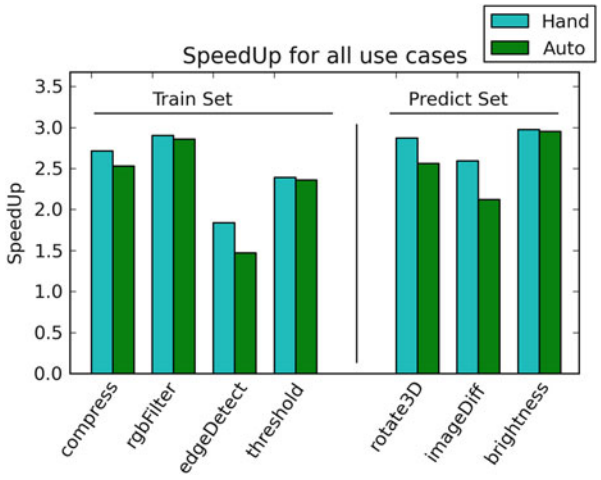

Fig. 19 Speedups for training and prediction sets

7 Conclusions and Future Work

We have presented a transformation toolchain that uses guarded rewriting rules and semantic information contained in annotations (which, together, make STML) in the source code to adapt initial code to different platforms. An engine that interprets and executes these rules plus a machine learning-based module that decides which rules have to be executed have been implemented. A preliminary evaluation with representative small to mid-sized examples suggests that this is a promising technique that can generate code with good performance results—at least on a par with what a seasoned human programmer can write.

As part of the plans for the future, we seek to improve STML and enhance and adapt Cetus to obtain more advanced / specific properties. At the same time, we are evaluating other analysis tools that can hopefully infer more precise information and for a wider range of code. On the one hand, we are exploring tools like PLuTo [3], PET [21], and the Clang / LLVM analyzers to dependency information in array-based loops. On the other hand, we are studying tools such as VeriFast [7] that can reason on dynamically-allocated mutable structures.

We plan to use additional benchmarks to train and evaluate the machine learning tool; that will likely need to enrich the feature vector used to generate program abstractions. We also plan to study the use of *multi-objective* rewards combining different properties. This would make it possible to define transformation strategies that, for example, could generate the code that consumes the least amount of energy among those with the shortest execution time. Finally, we want to explore the use of different learning rates for different states/transformation sequences in order to converge faster towards transformed codes.

Acknowledgements This work has been partially funded by EU FP7-ICT-2013.3.4 project 610686 *POLCA*, Comunidad de Madrid project S2013/ICE-2731 *N-Greens Software*, Generalitat Valenciana grant *APOSTD/2016/036* and MINECO Projects TIN2012-39391-C04-03/TIN2012-39391-C04-04 *StrongSoft*, TIN2013-44742-C4-1-R *CAVI-ROSE*, and TIN2015-67522-C3-1-R *TRACES*.

We are also grateful to the various members of the POLCA project consortium for many fruitful discussions and feedback. We are in particular indebted to Jan Kuper, Lutz Schubert, Daniel Rubio, Colin Glass, Lotfi Guedria, and Robert de Groote.

References

1. Agakov, F., et al.: Using machine learning to focus iterative optimization. In: Proceedings of the International Symposium on Code Generation and Optimization, CGO '06, pp. 295–305. IEEE Computer Society, Washington, DC (2006). doi:10.1109/CGO.2006.37
2. Bagge, O.S., Kalleberg, K.T., Visser, E., Haveraaen, M.: Design of the CodeBoost transformation system for domain-specific optimisation of C++ programs. In: Third International Workshop on Source Code Analysis and Manipulation (SCAM 2003), pp. 65–75. IEEE (2003). doi:10.1109/SCAM.2003.1238032
3. Bondhugula, U., Hartono, A., Ramanujam, J., Sadayappan, P.: A practical automatic polyhedral parallelizer and locality optimizer. SIGPLAN Not. **43**(6), 101–113 (2008). doi:10.1145/1379022.1375595
4. Danalis, A., et al.: The Scalable Heterogeneous Computing (SHOC) benchmark suite. In: Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units, pp. 63–74. ACM (2010). doi:10.1145/1735688.1735702
5. Dave, C., Bae, H., Min, S., Lee, S., Eigenmann, R., Midkiff, S.P.: Cetus: a source-to-source compiler infrastructure for multicores. IEEE Comput. **42**(11), 36–42 (2009). doi:10.1109/MC.2009.385
6. Huber, B.: The Language.C Package. https://hackage.haskell.org/package/language-c (2014)
7. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: Verifast: A powerful, sound, predictable, fast verifier for C and Java. In: Proceedings of the Third International Symposium on NASA Formal Methods, NFM 2011, Pasadena, CA, 18–20 April 2011, pp. 41–55 (2011). doi:10.1007/978-3-642-20398-5_4
8. Jacquard Computing Inc.: ROCCC 2.0 User's Manual, revision 0.74 edn. (2012). http://roccc.cs.ucr.edu/UserManual.pdf
9. Kaelbling, L.P., Littman, M.L., Moore, A.P.: Reinforcement learning: a survey. J. Artif. Intell. Res. **4**, 237–285 (1996). doi:10.1613/jair.301
10. Kuper, J., Schubert, L., Kempf, K., Glass, C., Bonilla, D.R., Carro, M.: Program transformations in the POLCA project. In: Giorgi, R., Silvano, C. (eds.) Proceedings of Design, Automation and Test in Europe (2016)
11. Lammel, R., Jones, S.P., Magalhaes, J.P.: The SYB Package. https://hackage.haskell.org/package/syb (2009)
12. Lindtjorn, O., Clapp, R.G., Pell, O., Fu, H., Flynn, M.J., Mencer, O.: Beyond traditional microprocessors for geoscience high-performance computing applications. IEEE Micro **31**(2), 41–49 (2011). doi:10.1109/MM.2011.17
13. Mariani, G., Palermo, G., Meeuws, R., Sima, V.M., Silvano, C., Bertels, K.: Druid: designing reconfigurable architectures with decision-making support. In: 19th Asia and South Pacific Design Automation Conference, Singapore, 20–23 January 2014, pp. 213–218 (2014). doi:10.1109/ASPDAC.2014.6742892
14. Marsland, S.: Machine Learning: An Algorithmic Perspective, 1st edn. Chapman & Hall/CRC, Boca Raton, FL (2009). doi:10.1111/j.1751-5823.2010.00118_11.x

15. Maxeler Technologies: Max Compiler MPT. https://www.maxeler.com/solutions/low-latency/maxcompilermpt/ (2016)
16. Pedregosa, F., et al.: Scikit-Learn: machine learning in Python. J. Mach. Learn. Res. **12**, 2825–2830 (2011)
17. Pekhimenko, G., Brown, A.: Efficient program compilation through machine learning techniques. In: Naono, K., Teranishi, K., Cavazos, J., Suda, R. (eds.) Software Automatic Tuning, pp. 335–351. Springer, New York (2010). doi:10.1007/978-1-4419-6935-4_19
18. Schaul, T., Bayer, J., Wierstra, D., Sun, Y., Felder, M., Sehnke, F., Rückstieß, T., Schmidhuber, J.: PyBrain. J. Mach. Learn. Res. (2010). doi:10.1145/1756006.1756030
19. Schupp, S., Gregor, D., Musser, D., Liu, S.M.: Semantic and behavioral library transformations. Inf. Softw. Technol. **44**(13), 797–810 (2002). doi:10.1016/S0950-5849(02)00122-2
20. Tamarit, S., Mariño, J., Vigueras, G., Carro, M.: Towards a semantics-aware code transformation toolchain for heterogeneous systems. In: Villanueva, A. (ed.) Proceedings of XIV Jornadas sobre Programación y Lenguajes (PROLE 2016), pp. 17–32 (2016). http://hdl.handle.net/11705/PROLE/2016/014
21. Verdoolaege, S., Grosser, T.: Polyhedral extraction tool. In: Second International Workshop on Polyhedral Compilation Techniques (IMPACT'12), Paris, pp. 1–16 (2012). http://impact.gforge.inria.fr/impact2012/workshop_IMPACT/verdoolaege.pdf
22. Visser, E.: Program transformation with Stratego/XT: rules, strategies, tools, and systems in StrategoXT-0.9. In: Lengauer, C., Batory, D., Consel, C., Odersky, M. (eds.) Domain-Specific Program Generation. Lecture Notes in Computer Science, vol. 3016, pp. 216–238. Springer (2004). doi:10.1007/978-3-540-25935-0_13