Interactive Source-to-Source Optimizations Validated using Static Resource Analysis

Guillaume Bertholon
Arthur Charguéraud
Thomas Kœhler
Begatim Bytyqi
Damien Rouhling
Inria & Université de Strasbourg, CNRS, ICube
Strasbourg, France

Abstract

Developments in hardware have delivered formidable computing power. Yet, the increased hardware complexity has made it a real challenge to develop software that exploits the hardware to its full potential. Numerous approaches have been explored to help programmers turn naive code into high-performance code, finely tuned for the targeted hardware. However, these approaches have inherent limitations, and it remains common practice for programmers seeking maximal performance to follow the tedious and error-prone route of writing optimized code by hand.

This paper presents OptiTrust, an interactive source-to-source optimization framework that operates on general-purpose C code. The programmer develops a script describing a series of code transformations. The framework provides continuous feedback in the form of human-readable *diff*s over conventional C code. OptiTrust supports advanced code transformations, including transformations exploited by the state-of-the-art DSL tools Halide and TVM, and transformations beyond the reach of existing tools. OptiTrust also supports user-defined transformations, as well as defining complex transformations by composition of simpler transformations. Crucially, to check the validity of code transformations, OptiTrust leverages a *static resource analysis* in a simplified form of Separation Logic. Starting from user-provided annotations on functions and loops, our analysis deduces precise resource usage throughout the code.

CCS Concepts: • **Software and its engineering** → **Software performance**; **Development frameworks and environments**; • **Theory of computation** → *Separation logic*.

Keywords: High performance code, Source-to-source optimization, Separation logic

ACM Reference Format:
Guillaume Bertholon, Arthur Charguéraud, Thomas Kœhler, Begatim Bytyqi, and Damien Rouhling. 2024. Interactive Source-to-Source Optimizations Validated using Static Resource Analysis. In *Proceedings of the 13th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis (SOAP '24), June 25, 2024, Copenhagen, Denmark*. ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3652588.3663320

1 Introduction

1.1 Motivation

Performance matters in numerous fields of computer science, and in particular in applications from machine learning, computer graphics, and numerical simulation. Massive speedups can be achieved by fine-tuning the code to best exploit the available hardware [15]. Between a naive implementation and an optimized implementation, it is common to see a speedup of the order of 50×—on a single core. For many applications, the code can then be accelerated further by one or two orders of magnitude by refining the code to exploit multicore parallelism or GPUs.

Yet, producing high performance code is hard. Over the past decades, nontrivial mechanisms with subtle interactions were integrated into hardware architectures. Reasoning about performance requires reasoning about the effects of multiple levels of caches, the limitations of memory bandwidth, the intricate rules of atomic operations, and the diversity of vector instructions (SIMD). These aspects and their interactions make it challenging to build cost models. For example, the cost of a memory access can range from one CPU cycle to hundreds of CPU cycles, depending on whether the corresponding data is already in cache. In the general case, accurately modeling cache behavior requires a deep understanding of the algorithm and hardware at play.

Accurately predicting runtime behavior is challenging for expert programmers, and appears beyond the capabilities of automated tools. Therefore, compilers struggle to navigate the exponentially large search space of all possible code

SOAP '24, June 25, 2024, Copenhagen, Denmark

candidates [24], resorting to best effort heuristics, and often failing to produce competitive code [3].

Today, it remains common practice in industry for programmers to write optimized code *by hand* [1, 9]. However, manual code optimization is unsatisfactory for at least three reasons. First, manually implementing optimized code is time-consuming. Second, the optimized code is hard to maintain through hardware and software evolutions. Third, the rewriting process is error-prone: not only every manual code edition might introduce a bug, but the code complexity also increases, especially when introducing parallelism. These three factors are exacerbated by the fact that optimizations typically make code size grow by an order of magnitude (for example, the optimized code for our following matrix multiplication case study is 7× bigger).

In summary, neither fully automatic nor fully manual approaches are satisfying for generating high performance code. Both machine automation and human insight are needed in the optimization process.

1.2 Contribution

This paper introduces OptiTrust, the first interactive optimization framework that operates on general-purpose C code and that supports and validates state-of-the-art optimizations. OptiTrust is open-source and available at the URL: https://github.com/charguer/optitrust.

In OptiTrust, the user starts from an unoptimized C code, and develops a *transformation script* describing a series of optimization steps. Each step consists of an invocation of a specific transformation at specified *targets*. OptiTrust provides an expressive target mechanism for describing, in a concise and robust manner, one or several code locations. On any step of the transformation script, the user can press a key shortcut to view the *diff* associated with that step, in the form of a comparison between two human-readable C programs. Concretely, a transformation script consists of an OCaml program linked against the OptiTrust library.

To ensure that the user applies only semantic-preserving transformations, OptiTrust performs validity checks that leverage our *static resource analysis*, which concretely takes the form of a type checking algorithm, in a type system featuring linear resources. This type system may be thought of as a variant of the Rust type system, or as a scaled down version of Separation Logic [20]. Our resource-based system aims to be similar in spirit to RefinedC [22], a Separation Logic-based type system for C code, even though we have not implemented all the features of RefinedC yet.

For type-checking resources, functions and loops need to be equipped with *contracts* describing their resource usage. These contracts may be inserted either directly as no-op annotations in the C source code, or they may be inserted by dedicated commands as part of the transformation script. OptiTrust is able to automatically infer simple loop contracts, thus not all loops need to be annotated manually. Every OptiTrust transformation takes care of updating contracts in order to reflect changes in the code. In other words, a well-typed program remains well-typed after a transformation.

Currently, OptiTrust only automates the application of transformations and the checking of their validity, but we also plan to explore future work to guide the user towards useful optimizations.

Next, we illustrate how OptiTrust works through an example optimization script.

```
void mm(float* C, float* A, float* B, int m, int n, int p) {
  __reads("A ⇝ Matrix2(m, p), B ⇝ Matrix2(p, n)");
  __modifies("C ⇝ Matrix2(m, n)");
  for (int i = 0; i < m; i++) {
    __xmodifies("for j in 0..n → &C[i][j] ⇝ Cell");
    for (int j = 0; j < n; j++) {
      __xmodifies("&C[i][j] ⇝ Cell");
      float sum = 0.0f;
      for (int k = 0; k < p; k++)
        sum += A[i][k] * B[k][j];
      C[i][j] = sum;
    }
  }
}
void mm1024(float* C, float* A, float* B) {
  __reads("A ⇝ Matrix2(1024, 1024), "
          "B ⇝ Matrix2(1024, 1024)");
  __modifies("C ⇝ Matrix2(1024, 1024)");
  mm(C, A, B, 1024, 1024, 1024);
}
```

Listing 1. Unoptimized matrix multiplication. The function `mm` multiplies the matrices `A` and `B` and stores the result in `C`. The function `mm1024` specializes input sizes to 1024. We write `A[i][k]` instead of `A[MINDEX2(m, p, i, k)]`, for conciseness. In the future, we plan to leverage a mechanism for automatically propagating size information.

2 OptiTrust by Example

In this section we present the features of OptiTrust through an example: optimizing matrix multiplication. The aim is to produce similar code as a reference TVM *schedule* that was written by an expert targeting Intel CPUs.[1] TVM is an industrial-strength domain-specific compiler for machine learning.

Annotated Code. We start from the C code presented in Listing 1: a naive, unoptimized implementation of matrix multiplication. To use OptiTrust, we annotate the code with resource contracts, which follow a double-underscore prefix.

The `mm` function reads a matrix `A` of size $m \times p$, reads a matrix `B` of size $p \times n$, and modifies a matrix `C` of size $m \times n$. This is explicitly described by the function contract and its `__reads` /`__modifies` clauses. Each clause mentions a set of resources

[1] https://tvm.apache.org/docs/how_to/optimize_operators/opt_gemm.html

```
!! Function.inline_def [cFunDef "mm"];
let tile (id, tile_size) = Loop.tile (int tile_size)
   ~index:("b" ^ id) ~bound:TileDivides [cFor id] in
!! List.iter tile [("i", 32); ("j", 32); ("k", 4)];
!! Loop.reorder_at ~order:["bi"; "bj"; "bk"; "i"; "k"; "j"]
   [cPlusEq ()];
!! Loop.hoist_expr ~dest:[tBefore; cFor "bi"] "pB"
   ~indep:["bi"; "i"] [cArrayRead "B"];
!! Matrix.stack_copy ~var:"sum" ~copy_var:"s" ~copy_dims:1
   [cFor ~body:[cPlusEq ()] "k"];
!! Omp.simd [cFor ~body:[cPlusEq ()] "j"];
!! Omp.parallel_for [cFunBody ""; cStrict; cFor ""];
!! Loop.unroll [cFor ~body:[cPlusEq ()] "k"];
```

Listing 2. OptiTrust script for optimizing `mm1024`.

separated by ",". For example, the resource `A ⇝ Matrix2(m, p)` specifies that the matrix at address `A` in memory has size $m \times p$ *and* gives permission to access this matrix: effectively the permission over every individual cell of the matrix, that is: `for i in 0..m → for j in 0..p → &A[i][j] ⇝ Cell`.

Each iteration of the loops with index `i` and `j` modifies a separate group of cells from matrix `C`. Loop contracts contain two kinds of clauses: clauses prefixed by "x" are used to describe resources *exclusive* to one iteration, and clauses prefixed by "s" to describe resources *shared* by all iterations. For example, in the loop with index `j`, the clause `__xmodifies("&C[i][j] ⇝ Cell")` indicates that only iteration `j` can modify the `j`-th cell of the `i`-th row of the matrix `C`. By contrast, the clause `__smodifies("for j in 0..n → &C[i][j] ⇝ Cell")` would have indicated that all loop iterations can modify the `n` cells of the `i`-th row of the matrix `C`. Similarly, `__xreads("&A[i][j] ⇝ Cell")` indicates that the `j`-th iteration is the only one that reads the `j`-th cell of `i`-th row of the matrix `A`, and `__sreads("A ⇝ Matrix2(m, p)")` indicates that every iteration may read any of the `m` × `p` cells of the matrix `A`. By default, resources are shared by all iterations, so in this code the following clause is inferred for the three loops: `__sreads("A ⇝ Matrix2(m, p), B ⇝ Matrix2(p, n)")`.

Transformation Script. To apply optimizations, we write an OptiTrust script in OCaml, as shown in Listing 2. For the reader not familiar with OCaml, `f x y` denote the call of `f` on the arguments `x` and `y`; the symbol `~` is used to provide optional (or named) arguments; `[x; y; z]` denotes a list; `(x, y, z)` denotes a tuple; `s1 ^ s2` denotes a string concatenation; and `let f x = e1 in e2` introduces a local function f. The transformation script calls a series of transformations that are functions taking various arguments, including a *target* as last argument.

The optimizations applied by this script improve data locality, parallelism, and specialize the matrix sizes. The script consists of 8 transformation steps, developed interactively: with the cursor on a line starting with `!!`, we can press (e.g.) "F6" in the VSCode editor to visualize the *diff* associated with the transformation on that line. All intermediate versions of the code consist of human-readable, executable C code. The `!!` operator is used purely to enable interactivity and early termination. Additionally, a complete transformation report can be generated and explored.[2]

Targets. As mentioned earlier, transformations take targets as parameters, that describe code locations. A target consists of a list of constraints (prefixed by "c") that is satisfied by code paths that go through nodes satisfying each constraint, in the given order. For example, `cFunDef "mm"` requires visiting a function definition with the name `"mm"`, and `cFor id` requires visiting a for loop over an index with the name `id`. Targets may also include special modifiers (the ones that make a target relative are prefixed by "t"). For example, `tBefore` allows targeting the interstice before an instruction. As another example, `cStrict` controls the depth: `[cFunBody ""; cStrict; cFor ""]` targets for loops over any index name that appear immediately within a function body with any name, as opposed to being nested within other constructs. Targets may also be given as arguments to constraints, for example, `cFor ~body:[cPlusEq ()] "k"` requires visiting a for loop over an index with the name `"k"`, whose body also contains a `+=` operation.

Transformations. The script from Listing 2 calls functions from the OptiTrust library called transformations. We use `Function.inline_def` to inline the definition of `mm` into the `mm1024` function that specializes $m = n = p = 1024$. We use `Loop.tile`, `Loop.reorder_at` and `Loop.hoist_expr` to apply loop transformations improving data locality and exposing new dimensions for parallelization. We are free to use any OCaml feature, here defining the local `tile` function to iteratively apply it over a list: tiling the loops over `i` and `j` by 32, and the loop over `k` by 4 to create outer loops with indices `bi`, `bj`, and `bk`. `Loop.hoist_expr` creates a new temporary matrix with name `pB` to store values of matrix `B` using a better layout, something that actually requires manually changing the reference code (the *algorithm*) in a tool like TVM. While the last target argument locates which expression to hoist, the `~dest` target argument additionally describes where to hoist it. We locally promote an array to the stack using `Matrix.stack_copy`, introducing efficient `memcpy` operations, and allowing for the use of SIMD vector registers. We use `simd` and `parallel_for` to add OpenMP pragmas for multithreading and vectorizing a number of the newly created loops. Finally, we unroll the loop over `k` to help the downstream C compiler recognize instruction-level parallelism.

Combined Transformations. As witnessed by the detailed report generated by OptiTrust[2], our relatively concise optimization script for matrix multiplication actually involves a fair number of *basic* transformation steps happening under the hood. For example, `Loop.reorder_at` is a

[2] https://files.inria.fr/optitrust/soap24/matmul.html

$$\overline{\begin{array}{l}\textbf{for}_{\ \gamma}\ i\ \textbf{in}\ r_i\ \{\\ \quad t_1; \Delta_1\\ \quad \Gamma_2\ t_2; \Delta_2\\ \}\end{array}} \longmapsto \begin{array}{l}\overline{\textbf{for}_{\ \gamma_1}\ i\ \textbf{in}\ r_i\ \{}\\ \quad t_1;\\ \}\\ \overline{\textbf{for}_{\ \gamma_2}\ i\ \textbf{in}\ r_i\ \{}\\ \quad t_2;\\ \}\end{array}$$

$$\textbf{correct if:} \begin{cases} i \text{ fresh in } \gamma.\text{shrd} \\ \gamma.\text{shrd.modifies} \cap \text{notRO}(\Delta_1) \cap \Delta_2 = \emptyset \\ \gamma.\text{shrd.modifies} \cap \text{notRO}(\Delta_2) \cap \Delta_1 = \emptyset \end{cases}$$

annotations produced on the right-hand-side:

$$\gamma_1 \equiv \begin{cases} \text{loop.vars} \equiv \gamma.\text{loop.vars} \\ \text{shrd} \equiv \gamma.\text{shrd} \cap \Delta_1 \\ \text{excl.pre} \equiv \gamma.\text{excl.pre} \\ \text{excl.post} \equiv R \end{cases} \qquad \gamma_2 \equiv \begin{cases} \text{loop.vars} \equiv \gamma.\text{loop.vars} \\ \text{shrd} \equiv \gamma.\text{shrd} \cap \Delta_2 \\ \text{excl.pre} \equiv R \\ \text{excl.post} \equiv \gamma.\text{excl.post} \end{cases}$$

$$R \equiv \text{cleanup}(\Gamma_2 - \gamma.\text{shrd})$$

Figure 1. `Loop.fission` transformation and its validation.

combined transformation that takes as argument a specific instruction, and takes a description of how we would like to order the loops around it. The transformation is defined recursively, "bringing down" the desired loops, from innermost to outermost. The call to `reorder_at` in our script involves 4 loop swaps, 6 loop fissions, and 2 hoist operations. In particular, the hoist operations result in turning the `sum` variable local to the inner loop into a 2D-array of values declared in an outer loop. The loop fissions isolate the initialization and the reads into this 2D-array into separate loop nests.

Typing Algorithm. While running the transformation script, every intermediate code is typechecked with our resource type system. Internally, function contract clauses such as `__reads` and `__modifies` that appear in the initial code are desugared into lower-level pre- and post- conditions. For linear resources, preconditions consume resources whereas postconditions produce resources. In this low-level representation, following standard separation logic, read-only permissions are encoded using *fractions* [6, 14]. Loop contracts are also desugared, into lower-level loop contracts (written γ). Low-level loop contracts separate resources in groups: γ.shrd.modifies and γ.shrd.reads describe resources modified and read by all iterations; γ.excl.pre and γ.excl.post describe resources that are exclusively consumed and produced by one iteration. Low-level loop contracts also bind logical variables in γ.loop.vars that the loop abstracts over (typically, the fraction variables of the read-only permissions).

The code is then typed according to the provided contracts by proceeding top to bottom, in a syntax-directed way that does not require difficult inferences. The typing context consists of the resources available at a given program point, where resources may be fully available, available in read-only mode (i.e., only a fraction is available) or available in "uninit" mode. Resources available in *uninit* mode cannot be read from before writing to them, which is useful to model when memory values for a resource are irrelevant.

To type a function body, the typing context is initialized with the precondition, and the final typing context is checked to imply the postcondition. Intuitively, the body of a function is given access to its consumed resources, and must return access to its produced resources. For every program point, our typing algorithm not only computes the resources available as typing contexts (written Γ), but also the local resource usage (written Δ).

Validity Checks. Leveraging the resource typing information, OptiTrust checks that each transformation applied by the script (Listing 2) preserves the semantics. The validity of a *combined* transformation is derived from the validity of all the *basic* transformations that it leverages. The validity of a *basic* transformation is verified by the OCaml implementation of that transformation, which is responsible for checking sufficient conditions under which it preserves semantics. To ensure that every intermediate code type-checks, each transformation must also maintain annotations such as those provided in the initial code (Listing 1). Simple examples are the `Loop.simd` and `Loop.parallel_for` transformations, that are correct if the annotations on the targeted loop captures the absence of interference: the γ.shrd.modifies resource set of the loop contract γ is empty.

A more complex example is the validity of `Loop.fission`, depicted in Figure 1. The transformation is described on the internal imperative λ-calculus representation of OptiTrust, where `for` loops are simplified to iterate over ranges. Annotations on the left (in green) represent resource information consumed by the transformation, and annotations on the right (in orange) represent resource information produced by the transformation.

Intuitively, loop fission is correct if the resources modified by t_1 and t_2 do not interfere across iterations. For γ.excl resources, there is no interference because each iteration is independent. For γ.shrd resources, we check for interference using Δ_1 and Δ_2: if t_1 modifies one resource from γ.shrd, then t_2 must not use this same resource; symmetrically, if t_2 modifies a resource, then t_1 must not use it. Note, however, that t_1 and t_2 may both read the same resource.

On top of checking for the correctness condition, the fission transformation must also synthesize loop contracts for the new loops, so that the resulting code still type-checks. For shrd resources, we simply project the subsets of γ.shrd resources used by t_1 and t_2. For excl resources, we preserve the previous pre- and post-conditions, but need to synthesize a new middle-point (R) corresponding to the iteration-exclusive resources available between t_1 and t_2. R is computed by

```
float* pB = (float*)malloc(sizeof(float[32][256][4][32]));
#pragma omp parallel for
for (int bj = 0; bj < 32; bj++) {
  for (int bk = 0; bk < 256; bk++) {
    for (int k = 0; k < 4; k++) {
      for (int j = 0; j < 32; j++) {
        pB[32768 * bj + 128 * bk + 32 * k + j] =
          B[1024 * (4 * bk + k) + 32 * bj + j]; }}}}
#pragma omp parallel for
for (int bi = 0; bi < 32; bi++) {
  for (int bj = 0; bj < 32; bj++) {
    float* sum = (float*)malloc(sizeof(float[32][32]));
    for (int i = 0; i < 32; i++) {
      for (int j = 0; j < 32; j++) {
        sum[32 * i + j] = 0.; }}
    for (int bk = 0; bk < 256; bk++) {
      for (int i = 0; i < 32; i++) {
        float s[32];
        memcpy(s, &sum[32 * i], sizeof(float[32]));
#pragma omp simd
        for (int j = 0; j < 32; j++) { // this loop is for k = 0
          s[j] += A[1024 * (32 * bi + i) + 4 * bk + 0] *
                  pB[32768 * bj + 128 * bk + 32 * 0 + j]; }
        // [...] similar unrolling, not shown, for k = 1, 2, 3
        memcpy(&sum[32 * i], s, sizeof(float[32])); }}
    for (int i = 0; i < 32; i++) {
      for (int j = 0; j < 32; j++) {
        C[1024 * (32*bi + i) + 32*bj + j] = sum[32*i + j]; }}
// [...] free instructions, not shown
```

Listing 3. Optimized C code produced by the OptiTrust script for `mm1024`. This code has similar structure and achieves similar performance as the reference output of TVM.

subtracting the shared resources (γ.shrd) from the resources available between t_1 and t_2 (Γ_2), and for technical scoping reasons, performing a final "cleanup".

Final Optimized Code. Listing 3 shows the optimized C code produced by our script. First, we checked that this output code matches the structural optimizations from the reference TVM case study. Note that TVM directly targets LLVM IR, and does not produce easily readable C code.

Second, we checked the performance. We benchmarked our code against TVM's code on a 4-core Intel i7-8665U CPU with AVX2 support. Both codes have similar runtime, corresponding to a speedup of 150× over the naive code.[3]

3 Comparison to Related Work

Now that we have seen how OptiTrust works by example, let us introduce a number of qualitative properties, before reviewing related tools for semi-automatic code optimization and finally explaining why OptiTrust achieves a unique combination of features.

- **Generality**: How large is the domain of applicability of the tool? In particular, is it restricted to a domain-specific language?
- **Expressiveness**: How advanced are the code transformations supported by the tool? Is it possible to express state-of-the-art code optimizations?
- **Control**: How much control over the final code is given to the user by the tool? In particular, is there a monolithic code generation stage?
- **Feedback**: Does the tool provide easily readable intermediate code after each transformation?
- **Composability**: Is it possible to define transformations as the composition of existing transformations? Can transformations be higher-order, i.e., parameterized by other transformations?
- **Extensibility** of transformations: Does the tool facilitate defining custom transformations that are not expressible as the composition of built-in ones?
- **Trustworthiness**: Does the tool ensure that user-requested transformations preserve the semantics of the code? Can it moreover provide mechanized proofs?

3.1 Related Work

Halide [19] is an industrial-strength domain-specific compiler for image processing. Halide popularized the idea of separating an *algorithm* describing what to compute from a *schedule* describing how to optimize the computation. This separation makes it easy to try different schedules. TVM [8] is a tool directly inspired by Halide, but tuned for applications to machine learning. Halide and TVM are inherently limited to their domain-specific languages. They do not support higher-order composition of transformations, and are not extensible [3, 18]. Moreover, understanding their output is difficult as the applied transformations are not detailed to the user. Interactive scheduling systems have been proposed to mitigate this difficulty [13].

Elevate [11] is a functional language for describing *optimization strategies* as composition of simple *rewrite rules*. Advanced optimizations from TVM and Halide can be reproduced using Elevate. One key benefit is extensibility: adding rewrite rules is much easier than changing complex and monolithic compilation passes [18]. Elevate strategies are applied on programs expressed in a functional array language named Rise, followed by compilation to imperative code. The use of a functional array language greatly simplifies rewriting, however it restricts applicability and makes controlling imperative aspects difficult (e.g. memory reuse).

Exo [12] is an imperative DSL embedded in Python, geared towards the development of high-performance libraries for specialized hardware. It is restricted to static control programs with linear integer arithmetic. Exo programs can be optimized by applying a series of source-to-source transformations. These transformations are described using a Python script, with simple string-based patterns for targeting code points. The user can add custom transformations, possibly defined by composition; higher-order composition seems possible but has not yet been demonstrated.

[3] We obtain a 90th percentile runtime of 9.4ms over 200 benchmark runs, and compare it to the 90th percentile of the naive code. Besides, the OptiTrust median runtime is slightly faster than the TVM median runtime.

Table 1. Overview of user-guided tools for high-performance code generation.

	Halide/TVM	Elevate+Rise	Exo	Clay/LoopOpt	ATL	Alpinist	Clava+LARA
Generality	◔	◐	◐	◐	◐	◔	●
Expressiveness	●	●	●	◐	◐	◔	◔
Control	◐	◐	◔	◐	◐	●	●
Feedback	◐	◐	●	●	◐	●	◐
Composability	○	●	◐	◐	●	○	●
Extensibility	○	●	●	○	●	●	●
Trustworthiness	◐	◐	◐	◐	●	●	○

Clay [2] is a framework to assist in the optimization of loop nests that can be described in the *polyhedral model* [10]. The polyhedral model only covers a specific class of loop transformations, with restriction over the code contained in the loop bodies, however it has proved extremely powerful for optimizing code falling in that fragment. Clay provides a decomposition of polyhedral optimizations as a sequence of basic transformations with integer arguments. The corresponding transformation script can then be customized by the programmer. Clint [26] adds visual manipulation of polyhedral schedules through interactive 2D diagrams. LoopOpt [7] provides an interactive interface that helps users design optimization sequences (featuring unrolling, tiling, interchange, and reverse of iteration order) that can be bound in a declarative fashion to loop nests satisfying specific patterns.

ATL [16] is a purely functional array language for expressing Halide-style programs. Its particularity is to be embedded into the Coq proof assistant. ATL programs can be transformed through the application of rewrite rules expressed as Coq theorems. With this approach, transformations are inherently accompanied by machine-checked proofs of correctness. The set of rules includes expressive transformations beyond the scope of Halide, and can be extended by the user. Once optimized, ATL programs are then compiled into imperative C code. Like Rise, generality and control are restricted by the functional array language nature of ATL.

Alpinist [21] is a *pragma*-based tool for optimizing GPU-level, array-based code, able to apply basic transformations such as loop tiling, loop unrolling, data prefetching, matrix linearization, and kernel fusion. The key characteristic of Alpinist is that it operates over code formally verified using the VerCors framework [5]. Concretely, Alpinist transforms not only the code but also its formal annotations. If Alpinist were to leverage transformation scripts instead of pragmas, it might be possible to chain and compose transformations; yet, this possibility remains to be demonstrated.

Clava [4] is a general-purpose C++ source-to-source analysis and transformation framework implemented in Java. The framework has been instantiated mainly for code instrumentation purpose and auto-tuning of parameters. Clava can also be used in conjunction with a DSL called LARA [23] for optimizing specific programs. LARA allows expressing user-guided transformations by combining declarative queries over the AST and imperative invocations of transformations, with the option to embed JavaScript code. The application paper on the Pegasus tool [17] illustrates this approach on loop tiling and interchange operations.

Table 1 summarizes the properties of the existing approaches, highlighting their diversity. The table is sorted by increasing generality. For the tools considered, this generality is negatively correlated with expressiveness, i.e., with how advanced the supported transformations are. Regarding generality, only Clava supports operating on general C code, yet provides absolutely no guarantees on semantics preservation. For each property considered, at least two tools show strengths on that property (above half score). However, even if we leave out the ambition of achieving mechanized proofs, each tool considered shows weaknesses on at least two properties (half score or less).

3.2 The Unique Features of OptiTrust

When considering the aforementioned criteria and tools, OptiTrust achieves a unique combination of features.

Generality. OptiTrust is generally applicable to optimizing C code. The code must parse using Clang, the parser of LLVM. The fragments of code that the user wishes to alter must moreover type-check in our resource type system. At the time of writing, we support only core features of the C language: sequences, loops, conditionals, functions, local and global variables, arrays, and structs. There is, however, no inherent limitation: OptiTrust could presumably be extended to support nearly all the C language (we do not plan to handle general goto's). Our resource type system currently only allows describing simple *shapes* of data structures, and does not yet allow specifying the stored values. That said, we have been planing to extend our implementation to a full-featured Separation Logic similar to RefinedC [22]. In summary, OptiTrust in its current form does not yet demonstrate full generality, however it has been designed towards that goal.

Expressiveness. The combination of three ingredients allows OptiTrust's users to generate their desired optimized code: (1) the use of a transformation script for describing a sequence of transformations; (2) the use of a *target* mechanism, allowing to precisely pinpoint where transformations should

be applied; (3) the availability of a catalog of general-purpose transformations, whose composition enables altering the code with a lot of flexibility.

Let us summarize the transformations currently supported in OptiTrust. For instruction-level transformations, we support: function inlining, constant propagation, instruction reordering, switching between stack and heap allocation, and basic arithmetic simplifications. For control-flow transformations, we support: loop interchange, loop tiling, loop fission, loop fusion, loop-invariant code motion, loop unrolling, loop deletion and loop splitting. For data layout transformations, we support: interchange of dimensions of an array, and array tiling. There are many more useful transformations for which we are working out sufficient correctness conditions.

Certain transformations may require nontrivial checks. For example, array tiling requires the tile size to divide the array size, and loop splitting requires arithmetic inequalities to hold. OptiTrust currently only validates simple conditions; in the future, more complex conditions could be handled using either SMT solvers or interactive theorem provers.

Control. Transformation scripts in OptiTrust empower the user with very fine-grained control over how the code should be transformed. A challenge is to allow for concise scripts. To that end, OptiTrust provides high-level *combined* transformations, effectively recipes for combining the *basic* transformations provided by OptiTrust. Section 2 presented the example of `Loop.reorder_at`, which attempts, using a combination of fission, hoist, and swap operations, to create a reordered loop nest around a specified instruction. Overall, the use of *combined* transformations allows for reasonably concise transformation scripts, with the user's intention being described at a relatively high level of abstraction. The user stays in control and can freely mix the use of concise abstractions and precise fine-tuning transformations.

Feedback. For each step in the transformation script, OptiTrust delivers feedback in the form of human-readable C code. The user usually only needs to read the *diff* against the previous code. Interestingly, OptiTrust also records a trace that allows investigating all the substeps triggered by a *combined* transformation. This information is critically useful when the result of a high-level transformation does not match the user's intention. Besides, a key feature of OptiTrust is its fast feedback loop. The production of fast, human-readable feedback in a system with significant control is reminiscent of interactive proof assistants, and of the aforementioned ATL tool [16].

Composability. OptiTrust transformation scripts are expressed as OCaml programs, and each transformation from our library consists of an OCaml function. Because OCaml is a full-featured programming language, OptiTrust users may define additional transformations at will by combining existing transformations. User-defined transformations may query the abstract syntax tree (AST) that describes the C code, allowing to perform analyses before deciding what transformations to apply. Furthermore, because OCaml is a higher-order programming language, transformation can take other transformations as argument. We use this programming pattern for example to customize the arithmetic simplifications to be performed after certain transformations.

Extensibility. If the user needs a transformation that is not expressible as a combination of transformations from the OptiTrust library, a custom transformation can be devised. Because OptiTrust does not rely on heuristics, adding a new transformation to OptiTrust does not impact in any way the behavior of existing scripts. To define relatively simple custom transformations, OptiTrust provides a term-rewriting facility based on pattern matching. For more complicated transformations, one can follow the patterns employed in the OptiTrust's library. For all custom transformations, it is the programmer's responsibility to work out the criteria under which applying the transformation preserves the semantics of the code, and to adapt contracts if necessary in order to produce well-typed code.

Trustworthiness. Compilers are well-known to be incredibly hard to get 100% correct [25]. Like compilers, optimization tools are highly subject to bugs. OptiTrust mitigates the risks of producing incorrect code in two ways.

Firstly, we instrumented OptiTrust to generate *reports* when processing transformation scripts. A report takes the form of a standalone HTML page, which contains the diff for every transformation step (and sub-steps). Such a report can be thoroughly scrutinized by a third-party reviewer.

Secondly, we have organized the OptiTrust code base so as to isolate the implementation of the *basic* transformations, which consists of transformations that directly modify the AST. Only basic transformations need to be trusted. We have been careful to systematically minimize the complexity of the interface and of the implementation of our basic transformations. All other transformations—the *combined* transformations—are *not* part of the trusted computing base.

4 Conclusion

To conclude, OptiTrust is general-purpose, takes as input C code, supports interactive development of transformation scripts, and produces at every step readable C code semantically equivalent to the original code. Our case study demonstrates that a reasonably concise OptiTrust script achieves the same performance as an expert-written TVM schedule.

In future work, we plan to integrate support for arbitrary logical assertions. Following the approach of Alpinist [21], OptiTrust will transform not only code, but also the expressive logical annotations that decorate the code. Leveraging on support for logical assertions, we look forward to completing more challenging case studies.

References

[1] Vasco Amaral, Beatriz Norberto, Miguel Goulão, Marco Aldinucci, Siegfried Benkner, Andrea Bracciali, Paulo Carreira, Edgars Celms, Luís Correia, Clemens Grelck, Helen Karatza, Christoph Kessler, Peter Kilpatrick, Hugo Martiniano, Ilias Mavridis, Sabri Pllana, Ana Respício, José Simão, Luís Veiga, and Ari Visa. 2020. Programming languages for data-Intensive HPC applications: A systematic mapping study. *Parallel Comput.* 91 (2020), 102584. https://doi.org/10.1016/j.parco.2019.102584

[2] Lénaïc Bagnères, Oleksandr Zinenko, Stéphane Huot, and Cédric Bastoul. 2016. Opening polyhedral compiler's black box. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization* (Barcelona, Spain) *(CGO '16)*. Association for Computing Machinery, New York, NY, USA, 128–138. https://doi.org/10.1145/2854038.2854048

[3] Paul Barham and Michael Isard. 2019. Machine Learning Systems are Stuck in a Rut. In *Proceedings of the Workshop on Hot Topics in Operating Systems* (Bertinoro, Italy) *(HotOS '19)*. Association for Computing Machinery, New York, NY, USA, 177–183. https://doi.org/10.1145/3317550.3321441

[4] João Bispo and João M. P. Cardoso. 2020. Clava: C/C++ source-to-source compilation using LARA. *SoftwareX* 12 (2020), 100565. https://doi.org/10.1016/J.SOFTX.2020.100565

[5] Stefan Blom, Saeed Darabi, Marieke Huisman, and Wytse Oortwijn. 2017. The VerCors Tool Set: Verification of Parallel and Concurrent Software. In *Integrated Formal Methods - 13th International Conference, IFM 2017, Turin, Italy, September 20-22, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10510)*, Nadia Polikarpova and Steve A. Schneider (Eds.). Springer, 102–110. https://doi.org/10.1007/978-3-319-66845-1_7

[6] John Boyland. 2003. Checking Interference with Fractional Permissions. In *Static Analysis, 10th International Symposium, SAS 2003, San Diego, CA, USA, June 11-13, 2003, Proceedings (Lecture Notes in Computer Science, Vol. 2694)*, Radhia Cousot (Ed.). Springer, 55–72. https://doi.org/10.1007/3-540-44898-5_4

[7] Lorenzo Chelini, Martin Kong, Tobias Grosser, and Henk Corporaal. 2021. LoopOpt: Declarative Transformations Made Easy. In *Proceedings of the 24th International Workshop on Software and Compilers for Embedded Systems* (Eindhoven, Netherlands) *(SCOPES '21)*. Association for Computing Machinery, New York, NY, USA, 11–16. https://doi.org/10.1145/3493229.3493301

[8] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Q. Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, Andrea C. Arpaci-Dusseau and Geoff Voelker (Eds.). USENIX Association, 578–594. https://www.usenix.org/conference/osdi18/presentation/chen

[9] Thomas M Evans, Andrew Siegel, Erik W Draeger, Jack Deslippe, Marianne M Francois, Timothy C Germann, William E Hart, and Daniel F Martin. 2022. A survey of software implementations used by application codes in the Exascale Computing Project. *The International Journal of High Performance Computing Applications* 36, 1 (2022), 5–12. https://doi.org/10.1177/10943420211028940

[10] Paul Feautrier. 1992. Some efficient solutions to the affine scheduling problem: one dimensional time. *Intl. Journal of Parallel Programming* 21, 5 (october 1992), 313–348. https://doi.org/10.1007/BF01407835

[11] Bastian Hagedorn, Johannes Lenfers, Thomas Kundefinedhler, Xueying Qin, Sergei Gorlatch, and Michel Steuwer. 2020. Achieving high-performance the functional way: a functional pearl on expressing high-performance optimizations as rewrite strategies. *Proc. ACM Program. Lang.* 4, ICFP, Article 92 (aug 2020), 29 pages. https://doi.org/10.1145/3408974

[12] Yuka Ikarashi, Gilbert Louis Bernstein, Alex Reinking, Hasan Genc, and Jonathan Ragan-Kelley. 2022. Exocompilation for productive programming of hardware accelerators. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 703–718. https://doi.org/10.1145/3519939.3523446

[13] Yuka Ikarashi, Jonathan Ragan-Kelley, Tsukasa Fukusato, Jun Kato, and Takeo Igarashi. 2021. Guided Optimization for Image Processing Pipelines. In *2021 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 1–5. https://doi.org/10.1109/VL/HCC51201.2021.9576341

[14] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *JFP* 28 (2018). https://doi.org/10.1017/S0956796818000151

[15] Vasilios Kelefouras and Georgios Keramidas. 2022. Design and Implementation of 2D Convolution on x86/x64 Processors. *IEEE Transactions on Parallel and Distributed Systems* 33, 12 (2022), 3800–3815. https://doi.org/10.1109/TPDS.2022.3171471

[16] Amanda Liu, Gilbert Louis Bernstein, Adam Chlipala, and Jonathan Ragan-Kelley. 2022. Verified Tensor-Program Optimization via High-Level Scheduling Rewrites. 6, POPL, Article 55 (jan 2022), 28 pages. https://doi.org/10.1145/3498717

[17] Pedro Pinto, João Bispo, João M. P. Cardoso, Jorge G. Barbosa, Davide Gadioli, Gianluca Palermo, Jan Martinovič, Martin Golasowski, Kateřina Slaninová, Radim Cmar, and Cristina Silvano. 2022. Pegasus: Performance Engineering for Software Applications Targeting HPC Systems. *IEEE Transactions on Software Engineering* 48, 3 (2022), 732–754. https://doi.org/10.1109/TSE.2020.3001257

[18] Jonathan Ragan-Kelley. 2023. Technical Perspective: Reconsidering the Design of User-Schedulable Languages. *Commun. ACM* 66, 3 (feb 2023), 88. https://doi.org/10.1145/3580370

[19] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Conference on Programming Language Design and Implementation*. 12 pages. https://doi.org/10.1145/2491956.2462176

[20] J.C. Reynolds. 2002. Separation logic: a logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*. 55–74. https://doi.org/10.1109/LICS.2002.1029817

[21] Ömer Sakar, Mohsen Safari, Marieke Huisman, and Anton Wijs. 2022. Alpinist: An Annotation-Aware GPU Program Optimizer. In *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 13244)*, Dana Fisman and Grigore Rosu (Eds.). Springer, 332–352. https://doi.org/10.1007/978-3-030-99527-0_18

[22] Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. 2021. RefinedC: automating the foundational verification of C code with refined ownership types. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 158–174. https://doi.org/10.1145/3453483.3454036

[23] Cristina Silvano, Giovanni Agosta, Andrea Bartolini, Andrea R. Beccari, Luca Benini, Loïc Besnard, João Bispo, Radim Cmar, João M.P. Cardoso, Carlo Cavazzoni, Daniele Cesarini, Stefano Cherubin, Federico Ficarelli, Davide Gadioli, Martin Golasowski, Antonio Libri, Jan Martinovič, Gianluca Palermo, Pedro Pinto, Erven Rohou, Kateřina Slaninová, and Emanuele Vitali. 2019. The ANTAREX domain specific language for high performance computing. *Microprocessors and Microsystems* 68 (2019), 58–73. https://doi.org/10.1016/j.micpro.2019.05.005

[24] Manish Vachharajani, Neil Vachharajani, David I. August, and Spyridon Triantafyllis. 2003. Compiler Optimization-Space Exploration. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE Computer Society, Los Alamitos, CA, USA, 204. https://doi.org/10.1109/CGO.2003.1191546

[25] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Conference on Programming Language Design and Implementation* (San Jose, California, USA). Association for Computing Machinery, 12 pages. https://doi.org/10.1145/1993498.1993532

[26] Oleksandr Zinenko, Stéphane Huot, and Cédric Bastoul. 2018. Visual Program Manipulation in the Polyhedral Model. *ACM Trans. Archit. Code Optim.* 15, 1, Article 16 (mar 2018), 25 pages. https://doi.org/10.1145/3177961

Received 01-MAR-2024; accepted 2023-04-19