In Search of a Program Generator to Implement Generic Transformations for High-performance Computing

Albert Cohen [1] Sébastien Donadio [1] Maria-Jesus Garzaran [2]
Christoph Herrmann [3] David Padua [2]

[1] ALCHEMY group, INRIA Futurs, Orsay, France
[2] DCS, University of Illinois at Urbana-Champaign
[3] FMI, University of Passau, Germany

ABSTRACT

The quality of compiler-optimized code for high-performance applications lags way behind what optimization and domain experts can achieve by hand. This paper explores in-between solutions, besides fully automatic and fully-manual code optimization. This work discusses how generative approaches can help the design and optimization of supercomputing applications. It outlines early results and research directions, using MetaOCaml for the design of a generative tool-box to design portable optimized code. We also identify some limitations of the MetaOCaml system. We finally advocate for an offshoring approach to bring high-level and safe metaprogramming to imperative languages.

1. INTRODUCTION AND MOTIVATION

High-performance computing seems to be turning into a mature domain. Architecture designs, programming languages, compilation and optimization techniques have lately been rather evolutionary. However, programmers for high-performance applications still complain about the lack of efficiency of their machines — the ratio of sustained to peak performance — and the poor performance of their optimizing or parallelizing compiler [40]. Indeed, the path from research prototypes to production-quality optimizers has been more difficult than expected, and advanced loop-nest and interprocedural optimizations are still performed manually by application programmers. The main reasons are the following:

- driving and selecting profitable optimizations becomes more and more difficult, due to the complexity and dynamic behavior of modern processors [24, 11, 30];
- domain-specific knowledge unavailable to the compiler can be required to prove optimizations' legality or profitability [6, 26];
- hard-to-drive transformations are not available in compilers, including transformations whose profitability is difficult to assess or whose risk of degrading performance is high, e.g., speculative optimizations [2, 34];
- complex loop transformations do not compose well, due to syntactical constraints and code size increase [10];
- some optimizations are in fact algorithm replacements, where the selection of the most appropriate code may depend on the architecture and input data [27].

Manual optimizations degrade portability: the performance of a C or Fortran code on a given platform does not preclude its performance on different architectures; not speaking about the use of low level, machine specific extensions. Several works have successfully addressed this issue, not by improving the compiler, but through the design of application-specific program generators, a.k.a. active libraries [39]. Such generators often rely on feedback-directed optimization to select the best generation strategy [35], but not exclusively [42]. The most popular examples are ATLAS [40] for dense matrix operations and FFTW [17] for the fast Fourier transform. Such generators follow an iterative optimization scheme, as depicted in Figure 1 in the case of ATLAS: an external control loop generates multiple versions of the optimized code, varying the optimization parameters, and an empirical search engine drives the search for the best combination of parameters. Most program transformations applied in these generators have been previously proposed for classical compiler frameworks, but existing compilers fail to apply them for the abovementioned reasons. Conversely, optimizations often involve domain-specific knowledge, from the specialization and interprocedural optimization of library functions [13, 9] to application-specific optimizations like algorithm selection [27]. Recently, the SPIRAL project [33] investigated a domain-specific extension of such program generators, operating on a domain-specific language of digital signal processing formulas. This project is one step forward to bridge the gap between application-specific generators and generic compiler-based approaches, and to improve the portability of application performance.

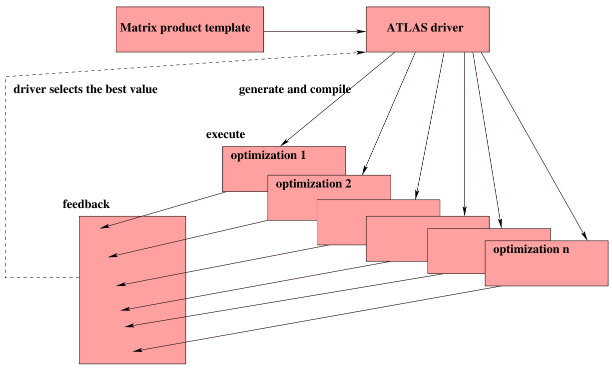

Figure 1: Empirical search in ATLAS

In this context, iterative compiler optimization is gaining more and more interest for its capability to drive complex transformations. It uses the feedback from real executions of the optimized program to explore the optimization search space using operations research algorithms [24], machine learning [27], and empirical experience [30]. In theory, iterative optimization could even be applied to domain-specific and user-defined transformations, if a framework for describing these transformations was available to application programmers. This is one of the goals of telescoping languages [22, 9], a compiler approach to reduce the overhead of calling generic library functions and to enable aggressive interprocedural optimizations, by lifting the semantical information about these libraries to the compiler. Beyond libraries, multiple alternative ideas have been proposed for domain-specific optimizations [26]. All these works highlight the increased need for researchers and developers in the field of high-performance computing to metaprogram their optimizations in a portable fashion.

In the multistage language and metaprogramming world, the most popular optimizations are partial evaluation, specialization and simplification. These transformations are useful to supercomputing applications but far from sufficient. As a matter of fact, research on generative programming and multistage evaluation has not greatly influenced the design of high-performance applications and compilers, most of the abovementioned projects being ad-hoc string-based program generators.

This paper discusses important issues raised by the application of generative approaches for high-performance computing. It outlines research directions and early results, using MetaOCaml for the design of a generative tool-box to design portable optimized code. In this context, we also explore how static type checking restricts the expressiveness of multistage evaluation. We finally present a heterogeneous multistage approach for safe metaprogramming in imperative languages.

2. GENERATIVE STRATEGIES FOR LOOP TRANSFORMATIONS

The main program transformations for high-performance target regular loop-oriented computations over arrays. Most transformations have been defined for an imperative, intraprocedural setting. Although more and more interprocedural program analyses are integrated into some modern compilers, few advanced interprocedural optimizations have been proposed. Advanced optimizations often fail to be applied by the compiler, if at all implemented, and some important optimizations would require too much domain specific knowledge or miss important hidden information (in libraries or input data structure) [9].

In a compiler perspective, there seem to be two means to improve the situation: either the programmer is given a way to teach the compiler new analyses and optimizations, how to drive them, and possibly when to apply them (overriding static analysis), or the programmer implements a generator for a class of programs to produce the optimized code automatically [26]. Multistage evaluation is primarily supporting the second direction and we will survey how typical transformations can be revisited in a multistage setting.

Very close to our work, the TaskGraph active library [5] provides multistage evaluation and loop transformation support for adaptive optimization of high performance codes (numerical and image processing, in particular). We share similar goals with this work, but we impose an additional constraint on the metaprogramming that may occur during program generation: *we aim for a purely generative approach, where code is only produced through multistage evaluation* (brackets, escape, run), whereas the TaskGraph approach implements loop transformations as high-level *transformations* of an abstract code representation. As a consequence, the TaskGraph library embeds a full restructuring compiler in the generator, where we only require back-end code generation. In addition, allowing code transformations in the metaprogram opens the door to unpredictable failures to apply the optimizations, due to pattern mismatches or syntactic limitations. This will even happen on an abstract representation like the TaskGraph IR [5]. Avoiding this unpredictability is a major reason for the programmer to choose to generate the optimized code instead of relying on an external compiler, and we will thus stick with a purely generative framework in the following.

```
let rec full_unroll lb ub body =
  if lb>ub then .< () >.
  else if lb=ub then body .< lb >.
  else .< begin .~(.~(body .< lb >.);
           full_unroll (lb+1) ub body) end >.

|val full_unroll :
|  int -> int -> (('a, int) code -> ('b, unit) code)
|  -> ('b, unit) code = <fun>

let a = Array.make 100 0

let body = fun i -> .< a.(.~i) <- .~i >.

full_unroll 0 3 body

|.< begin a.(0) <- 0; begin a.(1) <- 1; begin a.(2) <- 2;
      a.(3) <- 3 end; end; end >.
```

Figure 2: Full unrolling

2.1 Primitive Transformations

Let us consider a simple example: loop unrolling. Partial loop unrolling can be decomposed into a bound and stride recomputation step and a body duplication step. The second step is called full unrolling and is a straightforward application of multistage evaluation. The code in Figure 2 is a recursive multistage implementation of full loop unrolling; `body` is a function whose single argument is value of the loop counter for the loop being unrolled, and `lb` and `ub` are the loop bounds.[1] To enable substitution with an arbitrary expression, notice argument `i` is a code expression, not a plain integer. Although quite simple, implementing this transformation has two important caveats.

1. Free variables in code expressions are utterly important for code versioning, like for the specialization of `i` in this example. In MetaOCaml, all free variables must eventually be bound, e.g., as function arguments. Partial application handles practical cases where several generation steps with multiple specializations are needed.

2. The arguments of `full_unroll` and its return value have a different nature. It will not be easy to compose such a generation function with further transformation steps, unless subsequent transformations leave the duplicated loop bodies invariant.

[1] Vertical bars on the left delimit values returned by the toplevel.

The second issue is a major difficulty since transformation composition is key to practical optimization strategies for high performance. However, to achieve partial unrolling, we only have to convert the original loop bounds into adjusted bounds with a strided interval, then call `full_unroll` to perform the actual body duplication; see Figure 3.

```
let partial_unroll lb ub factor body =
  let number = (ub-lb)/factor in
  let bound = number*factor in
    .<
     begin
       for ii = 0 to number-1 do
         .~(full_unroll 0 (factor-1)
              (fun i -> body .< ii*factor+lb + .~i >.))
       done;
       for i = bound+lb to ub do
         .~(body .< i >.)
       done
     end
   >.

val partial_unroll :
  int -> int -> int -> (('a, int) code -> ('a, unit) code)
  -> ('a, unit) code = <fun>

let body = fun i -> .< a.(.~i) <- .~i >. in
  partial_unroll 1 14 4 body

.< begin
  for ii_2 = 0 to (3 - 1) do
    begin begin
      a.((ii_2 * 4) + 1) + 0 <- ((ii_2 * 4) + 1) + 0;
      a.((ii_2 * 4) + 1) + 1 <- ((ii_2 * 4) + 1) + 1 end;
      a.((ii_2 * 4) + 1) + 2 <- ((ii_2 * 4) + 1) + 2 end;
      a.((ii_2 * 4) + 1) + 3 <- ((ii_2 * 4) + 1) + 3
  done;
  for i_1 = (12 + 1) to 14 do
    a.((i_1) <- i_1
  done end >.
```

Figure 3: Partial unrolling

In the supercomputing context, this simple example and function inlining could be good advocates for multistage evaluation. However, neither inlining nor unrolling cause serious compilation problems today (beyond choosing when and how much to use them).

2.2 Complex Transformations

To experiment with a more realistic example, we reimplemented part of the ATLAS code generator for the matrix-matrix product [40, 42]. A pseudo-code for this generator is given in Figure 4; `NB` is the tile size, `MU` and `NU` are the unroll factors for the scalar-promoted block, `latency` is the number of iterations between the pipelined multiplications and additions in the inner products, `i1.(p)` and `i2.(p)` hold the automatically generated names for a large collection of scalar variables; the inner loops on `m` and `n` are fully unrolled and operate on these scalar variables. In addition to loop unrolling, three main transformations are involved.

Loop tiling. The outer loops of the matrix product are rescheduled to compute the blocked product for better cache locality [1]. This transformation is usually seen as a composition of strip-mining (making two nested loops out of one) and loop interchange.

None of these transformations makes much use of multistage evaluation. It is possible to implement bound and stride operations and loop generation for strip-mining in the same way as for partial unrolling, but this does not compose with loop interchange at the code expression level. Instead, we had to write an ad-hoc generator for the tiled template of the matrix product. We still get all the benefits of using MetaOCaml instead of strings or a syntax tree (as in ATLAS), but we do not enable any code reuse for further applications of loop tiling on other loop nests.

Scalar promotion. After a second tiling step, the innermost loops are fully unrolled (this is also called unroll and jam [1]) and array accesses in the large resulting blocks are promoted to scalars to enable register reuse (the whole transformation is called register tiling [1]).

This transformation should definitely fit a generative framework since it falls back to straightforward code substitution.

```
(* executed by the generator *)
h := 0;
for m=0 to MU-1 do
  for n=0 to NU-1 do
    i1.(h) = m;
    i2.(h) = n;
    h := !h+1
  done
done

(* Template of the generated code *)
for ii=0 to N-1 step NB do
  for jj=0 to M-1 step NB do
    for i=ii to ii+NB-NU step NU do
      for j=jj to jj+NB-MU step MU do
        (* Two fully unrolled loops *)
        for m=0 to MU-1 do
          for n=0 to NU-1 do
            c_m_n := c.(i+m).(j+n)
          done
        done;
        for k=0 to NB-1 do
          (* All fully unrolled loops *)
          for m=0 to MU-1 do
            a_m := a.(i+m).(k)
          done;
          for n=0 to NU-1 do
            b_n := b.(k).(j+n)
          done;
          for m=0 to latency-1 do
            t_i1.(m)_i2.(m) := !a_i1.(m) * !b_i2.(m)
          done;
          for m=0 to MU*NU-latency-1 do
            c_i1.(m)_i2.(m) := !c_i1.(m)_i2.(m)
                               + !t_i1.(m)_i2.(m);
            n := m + latency;
            t_i1.(n)_i2.(n) := !a_i1.(n) * !b_i2.(n)
          done;
          for m=MU*NU-latency to MU*NU-1 do
            c_i1.(m)_i2.(m) := !c_i1.(m)_i2.(m)
                               + !t_i1.(m)_i2.(m)
          done
        done;
        (* Two fully unrolled loops *)
        for m=0 to MU-1 do
          for n=0 to NU-1 do
            c.(i+m).(j+n) <- !c_m_n
          done
        done
      done
      (* Postlude... *)
    done
    (* Postlude... *)
  done
  (* Postlude... *)
done
(* Postlude... *)
```

Figure 4: Simplified matrix product template

However, one may not explicitely craft new variable names in MetaOCaml, since identifiers in `let` or `fun` bindings are not first-class citizens: `let . name = ...` is not a valid syntax. We know two methods to round this limitation. The first method assumes dynamic *single assignment* arrays [14]. Such arrays can be replaced by fresh scalar variables, whose names are automatically generated by the MetaOCaml system, following a monadic continuation-passing style [23]. The interested reader should refer to the latter paper for details. This approach has the advantage of directly generating efficient scalar code (unboxed), but it has most of the cons of programming with monads and explicit continuations, an unnatural style for programmers of high-performance numerical applications.

We studied another approach that does not assume the array is accessed in a *single assignment* manner [14], i.e., when an individual array element may be overwriten multiple times. The most natural solution is to use *code expressions of references of the scalar type*, and splicing the appropriate code expression in place of the original accesses to the array being scalar-promoted. This solution leads to efficient code, if the OCaml compiler unboxes the scalar references. This approach will be developed further in the paper.

Because none of these solutions is perfectly satisfactory, we will investigate how an offshoring approach — the safe generation of C code from MetaOCaml — can help.

Instruction scheduling. To better hide memory and floating point operation latencies, some instructions in the innermost loops are rescheduled in the loop body and possibly delayed or advanced by a few iterations.[2]

Instruction scheduling and software pipelining may not immediately seem as good cases for generative languages, but it actually fits in a typical scheduling strategy where instructions are extracted from a priority list and generated in order. It is even possible to write a generic list-scheduling algorithm [12] on a dependence graph of code expressions and to extend it to modulo-scheduling [28].

An important issue is code readability and debugging. Although static checking is a great tool for developing robust metaprograms, writing a generative template for the tiled, unrolled, scalar promoted and pipelined matrix-matrix product was often a trial and error game. Mixing functions on code expressions, escapes and partial application makes the code hard to follow, especially to supercomputing experts.

We did not investigate interprocedural optimizations beyond classical applications of generative languages, specialization, cloning and inlining (more complex transformations may even combine loop and function transformations [9]). We believe that addressing the issues raised by ATLAS will contribute to support such transformations as well.

We did not study the legality of the code transformations either. This issue is of course important, although not as critical as in a stand-alone compiler where the decision of considering a program transformation has to be fully automatic. If we were to check for array or scalar dependences [1] or to evaluate the global impact of data layout transformations, the OCaml type system would be insufficient. Coupling a multistage generator with a static analysis framework seems an interesting research direction.

[2]This optimization improves the effects of the scheduling heuristic in the backend compiler.

2.3 Prototyping Scalar Promotion

In this section, we use the meta-programming facilities of MetaOCaml to express the computation-expensive kernel of the program in a style in which array accesses are replaced by references to single variables. We expect that an advanced native-code compiler for MetaOCaml will use processor registers for the reference variables. The names for the reference variables are constructed internally by the MetaOCaml system (version 3.07) and are accessed through a list of code expressions, using the same indices as the original array.

We use the following three combinators:

- `withArray` takes the length of an array which should be turned into a collection of reference variables and a function which maps a list of these variables to a piece of code. This list can be taken as an environment. We refer to the variables by indexing this list, since their names (if printed) are not given by the MetaOCaml system before run time.
- `dynFor` creates a `for`-loop environment in which loops can have a stride greater than 1; it is based on OCaml `while`-loops. `dynFor` takes the lower bound, upper bound, stride and a function which abstracts the loop body in the iteration variable. All arguments of `dynFor` have to be code parts, since they depend on dynamic values.
- `staticFor` expresses loops of stride one which are to be fully unrolled in the program. `staticFor` takes a static lower and upper bound and a function for the loop body which maps the (static) index to the code part of the loop body.

Figure 5 shows the scalar-promoted loop program for the matrix multiplication without the body of the loop nest. The static arguments are the unrolling factors `mu` and `nu`, and the software pipeline `latency`. The dynamic arguments are listed in a lambda abstraction inside the code part which is nested in the innermost environment of the `withArray` applications. For each of the arrays `a`, `b`, `t` and `c` we apply the `withArray` combinator and define functions `aa`, `bb`, `tt` and `cc` which translate each access to an array into the access to the appropriate register.

The loop nest body is depicted in Figure 6. It consists of three parts. In the first part values of the result array `c` are loaded in the register set accessed by `cc`. The second part updates the values in these registers. In the third part, the values are stored back in the array. The second part consists of a dynamic loop on `k`, whose body consists of sequences of statements, which are expressed here in terms of our combinator `staticFor`. First, elements from the arrays `a` and `b` are loaded into the registers. The temporary register set accessed by `tt` is filled with initial products, then expresses the pipelining effect by concurrently being involved in multiplication and addition, and at last, when all products have been computed, is flushed by adding the pending values to the register set accessed by `cc`.

2.4 Composition of Transformations

```
let matmult mu nu latency =
  let i1 = Array.make (mu*nu) 0
  and i2 = Array.make (mu*nu) 0
  and h = ref 0 in
    for m=0 to mu-1 do
      for n=0 to nu-1 do
        i1.(!h) <- m;
        i2.(!h) <- n;
        h := !h+1
      done
    done;
    withArray mu (fun aregs -> let aa i   = nth aregs i in
    withArray nu (fun bregs -> let bb i   = nth bregs i in
    withArray (mu*nu) (fun tregs ->
      let tt i j = nth tregs (i*nu+j) in
    withArray (mu*nu) (fun cregs ->
      let cc i j = nth cregs (i*nu+j) in
      .< fun a b c nn mm nb ->
       .~(dynFor .<0>. .<nn-1>. .<nb>. .<fun ii ->
        .~(dynFor .<0>. .<mm-1>. .<nb>. .<fun jj ->
         .~(dynFor .<ii>. .<ii+nb-nu>. .<nu>. .<fun i ->
          .~(dynFor .<jj>. .<jj+nb-mu>. .<mu>. .<fun j ->
            LOOP NEST BODY
          >.) >.) >.) >.) >.))))
```

Figure 5: Scalar promotion — loop template

```
.~(staticFor 0 (mu-1) (fun m ->
     staticFor 0 (nu-1) (fun n ->
       .< .~(cc m n) := c.(i+m).(j+n) >.)));
for k=0 to nb-1 do
  .~(staticFor 0 (mu-1) (fun m ->
      .< .~(aa m) := a.(i+m).(k) >.));
  .~(staticFor 0 (nu-1) (fun n ->
      .< .~(bb n) := b.(k).(j+n) >.));
  .~(staticFor 0 (latency-1) (fun m ->
      .<         .~(tt (i1.(m)) (i2.(m)))
           := !(.~(aa (i1.(m))))
            * !(.~(bb (i2.(m)))) >.));
  .~(staticFor 0 (mu*nu-latency-1) (fun m -> let n = m+latency in
      .< begin
                 .~(cc (i1.(m)) (i2.(m)))
            := !(.~(cc (i1.(m)) (i2.(m))))
             + !(.~(tt (i1.(m)) (i2.(m))));
                 .~(tt (i1.(n)) (i2.(n)))
            := !(.~(aa (i1.(n))))
             * !(.~(bb (i2.(n))))
         end >.));
  .~(staticFor (mu*nu-latency) (mu*nu-1) (fun m ->
      .<           .~(cc (i1.(m)) (i2.(m)))
            := !(.~(cc (i1.(m)) (i2.(m))))
             + !(.~(tt (i1.(m)) (i2.(m)))) >.))
done;
.~(staticFor 0 (mu-1) (fun m ->
     staticFor 0 (nu-1) (fun n ->
       .< c.(i+m).(j+n) <- !(.~(cc m n)) >. )))
```

Figure 6: Scalar promotion — loop body

Figure 7 describes a real optimization sequence for the galgel SPEC CPU2000 FP benchmark [36] (borrowed from [30]). This sequence of 23 transformations on the *same program region* (a complex loop nest) was manually applied, following an optimization methodology for feedback-directed optimization [30]. The experimental platform is an HP AlphaServer ES45, 1 GHz Alpha 21264C EV68 with 8 MB L2 cache. Each analysis and transformation phase is depicted as a gray box, showing the time difference when executing the *full benchmark* (in seconds, a negative number is a performance improvement); the base execution time for each benchmark is 171 s. This sequence, although particularly complex, is representative of real optimizations performed by some (rare) compilers [20] and some (courageous) programmers [30]. Beyond compositionality, this example also shows how important are extensibility (provisions for implementing new transformations) and debugging support (static and/or generation-time and/or dynamic).

Figure 8 shows a practical example where 4 classical loop transformations convert the two simple loop nests above into the bloated code fragment below (partially shown): loop interchange, double loop fusion and software pipelining or shifting [1]. Multistage evaluation may clearly help the programmer to write a template for the code below, lifting several parameters and generation phases to automatically customize the code for a target architecture. Unfortunately, this template would not feature much code reuse for other loop optimizations.

Original nests.

```
for j = 1 to n do
  for i = 1 to m do
    a.(i) <- a.(i) + b.(i).(j) * c.(j)
  done
done;
for k = 1 to m do
  for l = 1 to n do
    d.(k) <- d.(k) + e.(l).(k) * c.(l)
  done
done
```

After 4 loop transformations.

```
let mn = min(m-4, n) in
  for x = 1 to mn do
    a.(1) <- a.(1) + b.(1).(x) * c.(x);
    a.(2) <- a.(2) + b.(2).(x) * c.(x);
    a.(3) <- a.(3) + b.(3).(x) * c.(x);
    a.(4) <- a.(4) + b.(4).(x) * c.(x);
    for y = 1 to mn do
      a.(y+4) <- a.(y+4) + b.(y+4).(x) * c.(x);
      d.(x) <- d.(x) + e.(y).(x) * c.(y)
    done;
    d.(x) <- d.(x) + e.(mn-3).(x) * c.(mn-3);
    d.(x) <- d.(x) + e.(mn-2).(x) * c.(mn-2);
    d.(x) <- d.(x) + e.(mn-1).(x) * c.(mn-1);
    d.(x) <- d.(x) + e.(mn).(x) * c.(mn);
    for y = mn+1 to m-4 do
      a.(y+4) <- a.(y+4) + b.(y+4).(x) * c.(x);
      d.(y) <- d.(y) + e.(y).(x) * c.(y)
    done;
    d.(m-3) <- d.(m-3) + e.(mn-3).(x) * c.(mn-3);
    d.(m-2) <- d.(m-2) + e.(mn-2).(x) * c.(mn-2);
    d.(m-1) <- d.(m-1) + e.(mn-1).(x) * c.(mn-1);
    d.(m) <- d.(m) + e.(mn).(x) * c.(mn)
  done;
  for x = mn+1 to n do
    for y = 1 to m-4 do
      ...
```

Figure 8: Composition of polyhedral transformations

The previous study makes clear that the lack of reusability limits the applicability of multistage evaluation for implementing advanced program transformations. Indeed, successful generative approaches should not only enable application programmers to implement a generator for one single program. In practice, code reuse and portability can be achieved using an abstract intermediate representation, from higher order skeletons — see e.g. [18, 19] — to more expressive domain-specific languages — see e.g. [37, 33] and the survey by Consel in [26].

Code expressions are rather constrained in terms of reusability, since code expressions produced by a program generator may not evolve beyond the predefined set of arguments of a function or the

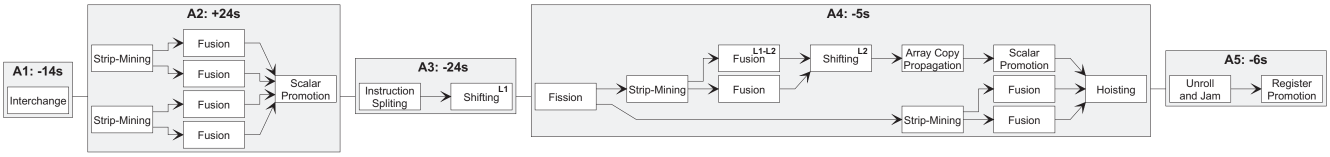

Figure 7: Optimizing galgel (base 171 s)

predefined escape points in the expression, as shown on the loop unrolling example. Any code fragment within a code expression is an invariant for further generation steps. As a result, many optimizations built of sequences of simpler transformations cannot be implemented as a composition of generators.

Let us study this limitation on loop unrolling again. Since partial loop unrolling can be decomposed into strip-mining and full unrolling of the inner strip-mined loop, we may implement the strip-mine generator shown in Figure 9, hoping that some composition mechanism will allow us to define partial unrolling from `strip_mine` and `full_unroll`. Unfortunately, this does not work as smoothly because `full_unroll` does not operate on a closed code expression but on a pair of integer arguments (the bounds) and a function on code expressions (the body).

```
let strip_mine lb ub factor body =
  let number = (ub-lb)/factor in
    .<
      begin
        for ii = 0 to number-1 do
          for i = 0 to factor-1 do
            .~(body .< ii*factor+lb+i >.)
          done
        done;
        for i = number*factor+lb to ub do
          .~(body .< i >.)
        done
      end
    >.

val strip_mine :
  int -> int -> (('a, int) code -> ('a, 'b) code)
  -> ('a, unit) code = <fun>

let body = fun i -> .< a.(.~i) <- .~i >. in
  strip_mine 1 14 4 body

.< begin
  for ii_8 = 0 to (3 - 1) do
    for i_9 = 0 to (4 - 1) do
      a.(((ii_8 * 4) + 1) + i_9) <- ((ii_8 * 4) + 1) + i_9
    done
  done;
  for i_7 = (12 + 1) to 14 do
    a.(i_7) <- i_7
  done end >.
```

Figure 9: Strip-mining

Now, combinators `strip_mine` and `partial_unroll` look very similar: let aside the recursive implementation of `full_unroll`, partial unrolling seems almost like a *lifted* version of strip-mining where the code expression has been extended from the loop body to the whole inner loop. Based on these similarities, it is indeed possible to *reuse* some code, by factoring the common template of `strip_mine` and `partial_unroll`. This is the purpose of the generalized strip-mining generator in Figure 10.[3] Thanks to the new `g_strip_mine` generator, it is easy to implement *both* the plain strip-mining and partial unrolling, by *composition*; see Figure 10 again. Here composition works thanks to code reuse opportunities; indeed, we did not overcome the fundamental asymmetry between the structured arguments of generator functions and the code expressions they produce.

```
let g_strip_mine gloop lb ub factor body =
  let number = (ub-lb)/factor in
    .<
      begin
        for ii = 0 to number-1 do
        .~(gloop 0 (factor-1)
            (fun i-> (body .< ii*factor+lb+ .~i >.)))
        done;
        for i = number*factor+lb to ub do
          .~(body .< i >.)
        done
      end
    >.

let g_loop_gen lw ub body =
  .< for i = lw to ub do .~(body .< i >.) done >.

(* Plain strip-mining *)
g_strip_mine g_loop_gen 1 14 4 body

(* Partial unrolling *)
g_strip_mine full_unroll 1 14 4 body
```

Figure 10: Factoring strip-mining and unrolling

Of course, an obvious circumvention to enable a compositional approach is to make the generators work on a suitable *intermediate representation*. The difference is then very thin compared with implementing the transformations in a compiler framework. Yet one may expect much benefits from the static checking and from an abstract view of the invariant code parts seen as polymorphic code expressions. For example, building an abstract intermediate representation for loop transformations is easy: the invariant pieces of code can be represented as code expressions, and the rest can be captured as trees of loop triplets.

After carrying these preliminary experiments, we do not believe a compositional approach based on multistage evaluation and abstract intermediate representations would be followed by application programmers. Indeed, the effort to master the language may not be worth the additional safety and abstraction. In addition, given the complexity and variety of program transformations for high-performance computing, most of the classical development errors will not even be avoided by such an approach (e.g., loop bounds or initialization inconsistencies). The next section further illustrate why better compositionality is needed to bring enough code reuse for the practical application of generative techniques to the design and optimization of supercomputing applications.

[3] Kindly provided by one of the reviewers of the original submitted paper.

2.5 Alternative Ideas

The polytope model has been proposed to represent loop nests and loop transformations in a unified fashion [16, 41, 21]. Although constrained by regularity assumptions on control structures, it is applicable to many loop nests [4] in scientific applications and captures loop-specific information as systems of affine inequalities. Recent advances in polyhedral code generation [3] and a new approach to enable compositional transformations [10] fit very well with our search for a generative and compositional optimization framework.

Fortunately, there exists a powerful OCaml interface [7] to the two most effective libraries to operate on polyhedra [15, 29, 32]. Together with multistage evaluation, it seems very easy and efficient to design a polyhedral code generator and to couple it with lower-level back-end optimizations, including scalar promotion and instruction scheduling. The use of the OCaml language should also facilitate the implementation of the symbolic legality-checking and profitability analyses involved in polyhedral techniques.

Besides the polytope model, alternative ways to achieve better compositionality may incur some "re-escaping" mechanism to perform further substitutions into closed code expressions. This may take the form of an extension of OCaml's pattern-matching construct, operating on elements of the OCaml grammar. Such an extension should not compromise the static checking properties of MetaOCaml, as long as type-invariant substitutions are performed, and some subtyping and specialization should also be possible. Unfortunately, this mechanism would not help overcoming the incompatibility between static checking and the multistage generation of new variable names, as required by scalar promotion (see the ATLAS template in Figure 4).

Of course, the ability to post-process generated code with pattern-based substitution is both a powerful tool... and a major infringement to the MetaOCaml design [8]. Although transformations operating at a more abstract level — like the polyhedral one — is undoubtedly more elegant, it is unlikely that a general-purpose user-extensible transformation framework will emerge anytime soon. Allowing some user control on the generated code is thus a pragmatic solution to consider.

3. SAFE METAPROGRAMMING IN C

There exist several two-stage evaluation extensions of the C language, from the standard C preprocessor to C++ template metaprogramming [38, 5], and to the 'C project [31] based on a fast run-time code generation framework. None of these tools provide the higher order functions on code expressions, cross-stage persistence and static checking features of MetaOCaml.

It seems possible to design a multistage C extension with most of these features, probably with strong restrictions on the language constructs allowed at the higher stages, but that would be a long-standing effort. In addition, OCaml is a language of choice to develop language processors. Using MetaOCaml as a generator for C programs seems more pragmatic, at least as a research demonstrator:

- this may either involve an OCaml to C translator, taking benefit of the imperative features of OCaml to ease the generation of efficient C code;
- or one may design a small preprocessor to embed C code fragments into the lower stages of a MetaOCaml program, relying on code generation primitives to assemble these fragments.

We advocate for the second direction, because it avoids transforming generated code, following the generative programming guidelines, and because it provides full C expressiveness with no hidden overhead. The rest of this section outlines interesting issues and presents simple solutions to this *offshoring* extension of MetaOCaml; it does not describe a realistic implementation and does not even aim at fully safe C code generation (yet).

3.1 Safe Generation of C code in OCaml

Our goal is to design a set of C code generation primitives such that *an OCaml program calling these primitives may only generate syntactically sound and type-safe C code*. We are not aware of any attempt to achieve this goal. Of course, from such generation primitives, it would be natural to design a preprocessor converting embedded C syntax into the proper OCaml declarations and calls.

Our current solution partially achieves this goal and imperfectly handles function declarations and calls. Practically, we mirror the C grammar productions as polymorphic generator functions operating on specific polyvariant types to represent meaningful C fragments. Each C variable is embedded into an OCaml pair of a string (its name) and a dummy value matching the type of the C variable, and these variables can only be declared through the explicit usage of OCaml variables. Figure 11 shows the main types and generation primitives (for intraprocedural constructs only) and Figure 12 shows a code generation example.

By design, the grammar productions guarantee syntax correctness. It is more challenging to deal with the scope of the C variable declarations and enforcing type-safety. We choose to represent a C block (the only placeholder for variable declarations) as a *function on an environment*. Interestingly, environments both record the code being generated and serve to delay the evaluation of the generation primitives until after the production of the surrounding C block syntax. This continuation passing style is the key to the embedding of C scoping and typing rules into the OCaml ones.

For example, an integer variable `x` is simultaneously declared to OCaml and generated to C code when evaluating

```
let x = gen_int_decl env "x".
```

Generated C code is appended to the environment `env`. Further assignments to `x` may call `gen_assign` directly and read references to `x` must first turn it into an "lvalue" through the `lvrv` function.

3.2 Generating New Names

Following this scheme, it is easy to support the generation of new names, without explicit usage of monads and continuation-passing style [23], and avoiding the possible overhead of the solution proposed in Section 2.3. Figure 13 shows a class to operate on scalar-promoted arrays with *generation-time* array bound checking. Of course, this checking cannot be done statically in general (although specific cases could be handled, with the proper encoding in the OCaml type system): scalar promoting an array with out-of-bound accesses yields out-of-bound exceptions.

For example, a scalar-promoted array `a` is simultaneously declared to OCaml and generated to C code when evaluating `let a = new`

```
type environment = {
  mutable txt:string; (* C text being produced *)
  mutable cnt:int; (* Private counter for alpha-renaming *)
  ind:string (* Pretty printing (indentation) *)
}
(* Some syntactic elements of C *)
type 'a c_rvalue = RValue of string * 'a
type 'a c_lvalue = LValue of string * 'a
type 'a c_loop = Cloop of ('a c_rvalue) * (bool c_rvalue)
                   * ('a c_rvalue) * (environment -> unit)

let get_nam var = match var with LValue (n, _) -> n
let get_def var = match var with LValue (_, d) -> d
let get_txt exp = match exp with RValue (t, _) -> t
let get_val exp = match exp with RValue (_, v) -> v
let get_init var = match var with Cloop (n, _, _, _) -> n
let get_cond var = match var with Cloop (_, c, _, _) -> c
let get_iter var = match var with Cloop (_, _, i, _) -> i
let get_body var = match var with Cloop (_, _, _, b) -> b

(* Support function: turns an lvalue into an rvalue *)
let lvrv var = RValue (get_nam var, get_def var)
(* Support function: build a loop object *)
let create_loop (init : 'a c_rvalue) (cond : bool c_rvalue)
  (iterator : 'a c_rvalue) block =
  Cloop (init, cond, iterator, block)

(* Output the declaration of <nam> to <env> *)
let gen_int_decl env nam =
  let var = nam ^ "_" ^ (string_of_int env.cnt) in
  let txt = sprintf "int %s;\n" var in
    env.txt <- env.txt ^ env.ind ^ txt;
    env.cnt <- env.cnt + 1;
    LValue (var, 0)
(* Output an instruction built of expression <exp> to <env> *)
let gen_inst env exp =
  let txt = sprintf "%s;\n" (get_txt exp) in
    env.txt <- env.txt ^ env.ind ^ txt

(* Output a block to <enclosing>
   Notice <block> is a function on environments, this allows
   to defer the evaluation of the generators in the block *)
let gen_block e block =
  let env = {txt=e.ind ^ "{\n"; cnt=0; ind=e.ind ^ "  "} in
    block env;
    env.txt <- env.txt ^ e.ind ^ "}\n\n";
    e.txt <- e.txt ^ env.txt

(* The following productions are self-explanatory *)

let gen_loop env loop  =
  env.txt <- env.ind ^ "for (" ^ (get_txt (get_init loop))
    ^ "; " ^ (get_txt (get_cond loop)) ^ "; "
    ^ (get_txt (get_iter loop)) ^ ")\n";
  gen_block env (get_body loop)
let gen_if_then_else env (cond : bool c_rvalue) b_then b_else =
  env.txt <- env.txt ^ "if (" ^ (get_txt cond) ^ ")\n";
  gen_block env b_then;
  env.txt <- env.txt ^ "else\n";
  gen_block env b_else
let gen_int_cst (cst : int) =
  let txt = sprintf "%d" cst in RValue (txt, cst)
let gen_assign (var : 'a c_lvalue) (exp : 'a c_rvalue) =
  let txt = sprintf "%s = %s" (get_nam var) (get_txt exp)
  in RValue (txt, exp)
let gen_lt (op1 : 'a c_rvalue) (op2 : 'a c_rvalue) =
  let txt = sprintf "(%s < %s)" (get_txt op1) (get_txt op2)
  in RValue (txt, true)
let gen_add (op1 : 'a c_rvalue) (op2 : 'a c_rvalue) =
  let txt = sprintf "(%s + %s)" (get_txt op1) (get_txt op2)
  in RValue (txt, (get_val op1) + (get_val op2))
```

Figure 11: Some generation primitives for C

```
let global = {txt=""; cnt=0; ind=""}

let generated_block =
  let block env =
    let x = gen_int_decl env "x"
    and y = gen_int_decl env "y" in
      gen_loop env
        (create_loop (gen_assign x (gen_int_cst 1))
          (gen_lt (lvrv x) (gen_int_cst 10))
          (gen_assign x (gen_add (lvrv x) (gen_int_cst 1)))
          (fun env ->
            gen_loop env
            (create_loop (gen_assign y (gen_int_cst 2))
              (gen_lt (lvrv y) (gen_int_cst 20))
              (gen_assign y (gen_add (lvrv y) (gen_int_cst 2)))
              (fun env ->
                 let z = gen_int_decl env "z" in
                   gen_inst env
                     (gen_assign z (gen_int_cst 5))))));
      gen_inst env
        (gen_assign y (gen_add (lvrv x) (gen_int_cst 3)));
      gen_if_then_else  env (gen_lt (gen_int_cst 2) (lvrv x))
        (fun env -> gen_inst env (gen_assign x (gen_int_cst 5)))
        (fun env -> gen_inst env (gen_assign x (gen_int_cst 3)))
  in gen_block global block

let _ =
  print_string global.txt
```

```
{
  int x_0;
  int y_2;
  for (x_0 = 1; (x_0 < 10); x_0 = (x_0 + 1))
  {
    for (y_2 = 2; (y_2 < 10); y_2 = (y_2 + 2))
    {
      int z_0;
      z_0 = 5;
    }
  }
  y_2 = (x_0 + 3);
  if ((2 < x_0))
  {
    x_0 = 5;
  }
  else
  {
    x_0 = 3;
  }
}
```

Figure 12: Example of C generation

```
class sp_int_array = fun nam siz ini env ->
  let _ = for i = 0 to siz-1 do
            env.txt <- env.txt ^ (sprintf "int %s_%d;\n" nam i)
          done in
object
  val name = nam
  val size = siz
  method idx x =
    if (0 <= x & x < siz)
      let var = sprintf "%s_%d" name x in
        LValue (var, 0)
    else
      failwith "Illegal reference to " ^ nam ^ "[" ^ x ^ "]"
end
```

Figure 13: Support for scalar-promotion

`sp_int_array "a" 10`, and it may further be referenced through the method `idx`.

3.3 Interprocedural Extensions

Currently, support for function declarations and calls is rather awkward. Like for scalar types, specific generator functions for each function type must be implemented. Three OCaml functions are needed per C function type: one for the function prototype, one for the declaration and one for the call. These generators operate on a polyvariant `'c_function` type and enforce the proper prototype/declaration/call ordering of the C language (assuming the generation of a single-file program). It is still unclear whether more polymorphism can be achieved, possibly requiring one triple of generators for each given number of arguments only.

The next step is of course to write the preprocessor for OCaml-embedded C code, to reduce the burden of writing code like the block generator in Figure 12.

3.4 Application to ATLAS

Figure 14 highlights the most significant parts of the implementation of a generator template for the matrix-matrix product. In this simple implementation, instruction scheduling is done by hand through an explicit split into three separate unrolled loops. Also, empirical search strategies to drive the optimization (e.g., simulated annealing) are not included.

```
(* ... *)

let q = new sp_int_array "q" 20 env
and r = new sp_int_array "r" 20 env
and t = new sp_int_array_array "t" 20 20 env

(* ... *)

in let block_mab1 env it =
  gen_block env
    (fun e -> gen_inst e (gen_assign (t#idx i1.(it) i2.(it))
      (gen_mul (lvrv (q#idx i1.(it))) (lvrv (r#idx i2.(it))))))

(* ... *)

in let multiply env =
  for m = 0 to mu-1 do
    for n = 0 to nu-1 do
      i1.(!h) <- m;
      i2.(!h) <- n;
      h := !h + 1
    done
  done;
  .! full_unroll 0 (latency - 1)
    (fun i -> .< block_mab1 env .~i >.);
  .! full_unroll 0 (mu * nu - latency - 1)
    (fun i -> .< block_mab2 env .~i >.);
  .! full_unroll (mu * nu - latency) (mu * nu - 1)
    (fun i -> .< block_mab3 env .~i >.)

(* ... *)
```

Figure 14: Revisited matrix product template

4. CONCLUSION AND PERSPECTIVES

We revisited some classical optimizations in the domain of high-performance computing, discussing the benefits and caveats of multistage evaluation. Since most advanced transformations are currently applied manually by domain and optimization experts, the potential benefit of generative approaches is very high. Indeed, several projects have followed ad-hoc string-based generation strategies to build adaptive libraries with self-tuned computation kernels. We took one of these projects as a running example. We show that MetaOCaml is suitable to implement this kind of application-specific generators, taking full advantage of its static checking properties. But our results also show that severe reusability and compositionality issues arise for complex optimizations, when multiple transformation steps are applied to a given code fragment. In addition, we believe that static type checking in MetaOCaml — in the current state of the system — complicates and somewhat restricts the usage of important optimizations like the scalar promotion of arrays.

Let us summarize our first results. Although multistage evaluation can be used in a simple and effective way to implement program generators for portable high-performance computing applications, it does not relieve the programmer from reimplementing the main generator parts for each target application. The only way to improve code reuse in the generator is to base its design on a custom intermediate representation, which may be almost as convoluted for application programmers as designing their own compiler.

In the second part of the paper, we addressed the execution overhead of writing high-performance applications in OCaml. We described a heterogeneous multistage approach to bring safe metaprogramming to C programs. By embedding all declarations of C variables into OCaml ones, it is possible to constrain the generator to produce syntactically correct and type-safe C code. Our first prototype shows that this approach can be combined with a front-end generator in MetaOCaml, enabling the design of a robust and elegant generator of safe optimized C code fully in MetaOCaml. Such an heterogeneous approach also lifts the name-generation restriction, but delays some safety checks to the generation of the C program.

In the future, we plan to use MetaOCaml as an experimental framework to evaluate multistage evaluation and possibly extend it to better support our adaptive optimization needs. We also wish to further investigate the potential of the heterogeneous MetaOCaml/C generator, combining it with a preprocessor to embed actual C code in MetaOCaml. In parallel, this work will help us design a metaprogramming layer on top of LLVM [25], a robust and extensible framework for lifelong program analysis and transformation.

Acknowledgments

This work is supported by exchange programs between the French CNRS, the University of Illinois, and the German DAAD. We wish to thank Paul Feautrier, Vikram Adve, Marc Snir, Changhao Jiang, Patrick Meredith, Xiaoming Li, Chris Lengauer and Walid Taha.

5. REFERENCES

[1] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan and Kaufman, 2002.

[2] D. I. August, D. A. Connors, S. A. Mahlke, J. W. Sias, K. M. Crozier, B.-C. Cheng, P. R. Eaton, Q. B. Olaniran, and W.-M. Hwu. Integrated predicated and speculative execution in the IMPACT EPIC architecture. In *Proceedings of the 25th Intl. Symp. on Computer Architecture*, July 1998.

[3] C. Bastoul. Efficient code generation for automatic parallelization and optimization. In *ISPDC'2 IEEE International Symposium on Parallel and Distributed Computing*, Ljubjana, Slovenia, Oct. 2003.

[4] C. Bastoul, A. Cohen, S. Girbal, S. Sharma, and O. Temam. Putting polyhedral loop transformations to work. In *Workshop on Languages and Compilers for Parallel Computing (LCPC'03)*, LNCS, College Station, Texas, Oct. 2003.

[5] O. Beckmann, A. Houghton, P. H. J. Kelly, and M. Mellor. Run-time code generation in c++ as a foundation for domain-specific optimisation. In *Proceedings of the 2003 Dagstuhl Workshop on Domain-Specific Program Generation*, 2003.

[6] A. J. C. Bik, M. Girkar, P. M. Grey, and X. Tian. Automatic intra-register vectorization for the intel architecture. *Int. J. of Parallel Programming*, 30(2):65–98, 2002.

[7] P. Boulet and X. Redon. SPPoC: Symbolic parameterized polyhedral calculator. `http://www.lifl.fr/west/sppoc`.

[8] C. Calcagno, W. Taha, L. Huang, and X. Leroy. Implementing multi-stage languages using asts, gensym, and reflection. In *ACM SIGPLAN/SIGSOFT Intl. Conf. Generative Programming and Component Engineering (GPCE'03)*, pages 57–76, 2003.

[9] A. Chauhan and K. Kennedy. Optimizing strategies for telescoping languages: procedure strength reduction and procedure vectorization. In *ACM Int. Conf. on Supercomputing (ICS'04)*, pages 92–101, June 2001.

[10] A. Cohen, S. Girbal, and O. Temam. A polyhedral approach to ease the composition of program transformations. In *EuroPar'04*, LNCS, Pisa, Italy, Aug. 2004. Springer-Verlag.

[11] K. D. Cooper, D. Subramanian, and L. Torczon. Adaptive optimizing compilers for the 21st century. *J. of Supercomputing*, 2002.

[12] A. Darte, Y. Robert, and F. Vivien. *Scheduling and Automatic Parallelization*. Birkhaüser, Boston, 2000.

[13] L. De Rose and D. Padua. Techniques for the translation of matlab programs into fortran 90. *ACM Trans. on Programming Languages and Systems*, 21(2):286–323, 1999.

[14] P. Feautrier. Array expansion. In *ACM Int. Conf. on Supercomputing*, pages 429–441, St. Malo, France, July 1988.

[15] P. Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22:243–268, Sept. 1988.

[16] P. Feautrier. Some efficient solutions to the affine scheduling problem, part II, multidimensional time. *Int. J. of Parallel Programming*, 21(6):389–420, Dec. 1992. See also Part I, one dimensional time, 21(5):315–348.

[17] M. Frigo and S. G. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proc. of the ICASSP Conf.*, volume 3, pages 1381–1384, 1998.

[18] C. A. Herrmann and C. Lengauer. Parallelization of divide-and-conquer by translation to nested loops. *J. of Functional Programming*, 9(3):279–310, 1999.

[19] C. A. Herrmann and C. Lengauer. HDC: A higher-order language for divide-and-conquer. *Parallel Processing Letters*, 10(2/3):239–250, 2000.

[20] KAP C/OpenMP for Tru64 UNIX and KAP DEC Fortran for Digital UNIX. `http://www.hp.com/techsevers/software/kap.html`.

[21] W. Kelly. Optimization within a unified transformation framework. Technical Report CS-TR-3725, University of Maryland, 1996.

[22] K. Kennedy. Telescoping languages: A compiler strategy for implementation of high-level domain-specific programming systems. In *Proc. Intl. Parallel and Distributed Processing Symposium (IPIPS'00)*, pages 297–304, 2000.

[23] O. Kiselyov, K. N. Swadi, and W. Taha. A methodology for generating verified combinarorial circuits. In *Embedded Software Conf. (EMSOFT'04)*, Pisa, Italy, Sept. 2004.

[24] T. Kisuki, P. Knijnenburg, M. O'Boyle, and H. Wijshoff. Iterative compilation in program optimization. In *Proc. CPC'10 (Compilers for Parallel Computers)*, pages 35–44, 2000.

[25] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *ACM Conf. on Code Generation and Optimization (CGO'04)*, San Jose, CA, Mar. 2004.

[26] C. Lengauer, D. Batory, C. Consel, and M. Odersky, editors. *Domain-Specific Program Generation*. Number 3016 in LNCS. Springer-Verlag, 2003.

[27] X. Li, M.-J. Garzaran, and D. Padua. A dynamically tuned sorting library. In *ACM Conf. on Code Generation and Optimization (CGO'04)*, San Jose, CA, Mar. 2004.

[28] J. Llosa. Swing modulo scheduling: A lifetime-sensitive approach. In *Parallel Architectures and Compilation Techniques (PACT'96)*, 1996.

[29] V. Loechner and D. Wilde. Parameterized polyhedra and their vertices. *Int. J. of Parallel Programming*, 25(6), Dec. 1997. `http://icps.u-strasbg.fr/PolyLib`.

[30] D. Parello, O. Temam, A. Cohen, and J.-M. Verdun. Towards a systematic, pragmatic and architecture-aware program optimization process for complex processors. In *ACM Supercomputing'04*, 2004. To appear.

[31] M. Poletto, W. C. Hsieh, D. R. Engler, and M. F. Kaashoek. 'C and tcc: A language and compiler for dynamic code generation. *ACM Trans. on Programming Languages and Systems*, 21(2):324–369, Mar. 1999.

[32] W. Pugh. A practical algorithm for exact array dependence analysis. *Communications of the ACM*, 35(8):27–47, Aug. 1992.

[33] M. Püschel, B. Singer, J. Xiong, J. Moura, J. Johnson, D. Padua, M. Veloso, and R. W. Johnson. SPIRAL: A generator for platform-adapted libraries of signal processing algorithms. *Journal of High Performance Computing and Applications, special issue on Automatic Performance Tuning*, 18(1):21–45, 2004.

[34] L. Rauchwerger and D. Padua. The LRPD test: Speculative run–time parallelization of loops with privatization and reduction parallelization. *IEEE Transactions on Parallel and Distributed Systems, Special Issue on Compilers and Languages for Parallel and Distributed Computers*, 10(2):160–180, 1999.

[35] M. D. Smith. Overcoming the challenges to feedback-directed optimization. In *ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization*, pages 1–11, 2000. (Keynote Talk).

[36] Standard performance evaluation corporation. `http://www.spec.org`.

[37] A. Van Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. *ACM SIGPLAN Notices*, 35(6):26–36, 2000.

[38] T. Veldhuizen. Using C++ template metaprograms. *C++ Report*, 7(4):36–43, 1995.

[39] T. Veldhuizen and D. Gannon. Active libraries: Rethinking the roles of compilers and libraries. In *SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing*, Oct. 1998.

[40] R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimizations of software and the atlas project. *Parallel Computing*, 2000.

[41] M. E. Wolf. *Improving Locality and Parallelism in Nested Loops*. PhD thesis, Stanford University, Aug. 1992. Published as CSL-TR-92-538.

[42] K. Yotov, X. Li, G. Ren, M. Cibulskis, G. DeJong, M. Garzaran, D. Padua, K. Pingali, P. Stodghill, and P. Wu. A comparison of empirical and model-driven optimization. In *ACM Symp. on Programming Language Design and Implementation (PLDI'03)*, San Diego, CA, June 2003.