On polynomial Code Generation

Paul Feautrier, Albert Cohen, Alain Darte

On Polynomial Code Generation

Paul Feautrier * Albert Cohen † Alain Darte ‡

December 18, 2018

Abstract

In static analysis, one often has to deal with polynomials in the program control variables, either native to the source code or created by enabling analyses. We have explained elsewhere how to compute dependences in such situations and use them for building polynomial schedules. It remains to explain how to generate polynomial code. The present proposal is to target new parallel programming languages of the async/finish family, like X10 or Habanero, which are "polynomial friendly" and for which efficient compilers exist.

Both these languages have barrier-like constructs—clocks for X10 and phasers for Habanero—which may be used to synchronize activities. To understand the behavior of a clocked program, one has to count the number of instances of clock `advance` instructions since the creation of each activity. Advance instructions with equal counts are synchronized, and these counts may be polynomials. The trick is therefore to insure that before executing a statement instance, its activity has executed as many `advance` instructions as the current value of its schedule. This can be obtained by inserting auxiliary loops for executing the necessary `advance` instructions.

This scheme fails if the schedule is not monotonically increasing with respect to the execution order in each activity. This problem may be solved by reordering the activities—which is possible since the real execution order is given by the schedule—or in extreme cases by index set splitting.

1 Context

Parallel programs need optimizing compilers capable of analyzing, transforming, and generating code with complex concurrent and parallel constructs. While polyhedral methods are popular means to reason about nested loops and data parallelism, the concurrency arising from task parallel and streaming languages

*Equipe INRIA Parkas and ENS, Lyon, France
†INRIA and ENS, Paris, France
‡Xilinx and ENS, Lyon, France

leads to more complex control flow and dependence patterns. One important family of transformations consists in converting one expression of parallelism into another, closer to the machine. For example, one may need to convert massive amounts of task-level concurrency in the stream language OpenStream [6] to data parallel execution or to "compile" this parallelism into coarser grain tasks. In the same spirit, an optimizing compiler for parallel programs may aim at reducing the dynamic range of runtime behavior by restricting the schedules to a constrained set of parallel executions, thereby controlling memory usage in streams, improving temporal locality, avoiding false sharing, bounding latency, etc. We argue that polynomial scheduling methods are well suited to express such transformations, and we investigate the down-stream schedule-guided generation of syntactic code for a target parallel language using clocks.

With this motivation in mind, let us review the main challenges and state of the art in polynomial methods used to reason about the transformation of task-parallel programs.

1.1 Why Polynomials?

The necessity of extending the polyhedral model beyond affine expressions and constraints has been around for some time. Obvious candidates for such extensions are polynomials and polynomial constraints, the so-called semi-algebraic sets replacing polyhedra. Polynomials may be native to the source code, or result from enabling analyses, like induction variable evaluation, message counting, array linearization, or become a way to schedule programs that cannot run in linear time. Taking advantage of recent mathematical results on the "PositivStellen Satz", a method to deal with polynomials has been proposed [10] and applied both to sequential programs and to a streaming language, OpenStream [6]. To pursue this line of research, it remains in particular to explain how to generate a parallel program from polynomial schedules.

1.2 An Overview of the OpenStream Language

The main components of an OpenStream program are streams, tasks, and task instances. A stream is a possibly unbounded one-dimensional array, with a read pointer and a write pointer. A task is an ordinary piece of C code, with statements for accessing streams. The reader is referred to [21] for a detailed description of the OpenStream language and run-time system.

1.2.1 The Control Program

The control program is a piece of sequential C code, expanded with task instance creation statements. When a task instance is created, it is given access to a subset of the control program variables, mainly the current values of the surrounding loop counters, and to some streams. The control program may execute arbitrary sequential code and transmit results to tasks via the `firstprivate` clause inherited from OpenMP. We will ignore this feature in this work as,

firstly, it increases the sequential fraction of the execution time and hence, by Ahmdal law, reduces the efficiency of the program, secondly because we found it possible to move these code fragments into tasks and use streams to distribute the results.

1.2.2 The Polyhedral Fragment of OpenStream

This work applies only to the case where the control program fits in the polyhedral model [6, 11]. The only executable statements are task creation statements, and the only control statements are the sequence and arbitrarily nested counted loops. Loop bounds must be affine forms in the surrounding loop counters and integer parameters. Many streaming applications—for instance, signal processing ones—deal with potentially infinite streams. These situations can be modeled using infinite loops, i.e., by omitting the termination test in the `for` construct.

Under these constraints, one may label task creation instances by an integer vector of the enclosing loop iterators in the control program—the familiar $2d+1$ notation—in such a way that their execution order or "happens before" relation, noted $\prec$ is simply the lexicographic order of their labels.

1.2.3 Streams and Tasks

Each task can access a finite set of *streams*. At instantiation time, the control program allocates a window into each accessible stream. Each stream s has a write pointer I_s, resp. a read pointer J_s, which specify the starting position of the current write window, resp. the current read window. For each stream access, the programmer specifies two nonnegative integers, the burst and the horizon. The horizon gives the size of the window. A null horizon is meaningless. The burst is the amount by which the read or write pointer is incremented after each access. The burst of a write access must be strictly positive and equal to the horizon, so as to insure single assignment. A read access may have a null burst: this implements the `peek` operation.

A task has a body: a piece of arbitrary code acting on the contents of its stream windows and local variables. A task is stateless: if transmitting information from one instance to another is needed, it must go through a stream or a network of streams. The present scheduler assumes that a task instance executes in unit time. It would be easy to assign to each task arbitrary delays, perhaps obtained from some WCET analysis, and take them into account in scheduling.

1.2.4 Stream Arrays

Streams can be organized into arrays of arbitrary size and dimension. Using the familiar C syntax, the name of a stream array must be subscripted by a fixed number of expressions. To allow static analysis, these subscripts must be affine forms or polynomials in the surrounding loop counters and parameters.

The original OpenStream compiler has been developed as an extension to `gcc`, a choice motivated by the ease of parallelizing and optimizing existing applications. Yet this choice is far from ideal to prototype polyhedral analyses and transformations. To simplify the present work, a more compact pseudo-code has been introduced in [6] and was extended as needs arose. This pseudo-code has C syntax for control operations, extended with declarations for parameters and streams, and executable statements for task creation. All examples in this paper use this pseudo-code; we hope the reader will find them self-explanatory.

1.3 Static Analysis

If the control program of an OpenStream program fits in the polyhedral model, one can statically analyze it up to scheduling and deadlock detection [6]. The first step is to obtain closed form expressions for the read and write pointers of each stream. Since at each task creation these pointers are incremented by the burst, it is enough to count the number of creations of relevant tasks which precede a given point in the execution of the control program. These creations can be labeled by integer points in union of parametric polyhedra, hence can be counted using the theory of Ehrhart polynomials [8, 3] or Brion's generating functions [2, 23] for which there exists efficient software.[1] The resulting counts are usually polynomials or piecewise polynomials of a degree bounded by the loop nesting level of the control program. In infrequent cases, integer division may occur in the resulting formulas. The burst is a numerical or parametric constant; the barvinok library is also able to compute weighted sums when the bursts are polynomials. However, in the presence of infinite loops, polynomial bursts must be used with care as a polynomial in the iterators of the control program is either constant or unbounded. The case where bursts are polynomials in the parameters is an interesting generalization of cyclo-static dataflow graphs where weights are parametric, pointing at a potentially important research direction for embedded applications.

Dependences Since write bursts are strictly positive and equal to the horizons, streams have the single assignment property. Hence, tasks have only flow dependences, when a task creates a datum that is used later by another task, or, in other words, when a write window and a read window overlap. Once the read and write pointers have been evaluated by barvinok, closed form formulas for dependences can be obtained. Since these formulas may use polynomials, the dependence relations are represented as semi-algebraic sets, and polynomial schedules can be obtained as in [10]. The polynomial scheduling problem is only semi-decidable: a failure does not necessarily imply the non existence of a polynomial schedule and hence a deadlock; it may also mean that the degree of the schedule has been underestimated. The present scheduler tries to construct bounded delay schedules, and rejects a program when no such schedule exists,

[1] The isl library: `http://isl.gforge.inria.fr`; and the barvinok library and `iscc`: `http://barvinok.gforge.inria.fr`.

on the ground that implementing unbounded delays needs unbounded memory.

2 A Simplified View of X10

The basic component of an X10 program [22] is the *activity*, a lightweight thread. An activity is created by the primitive `async{ <body> }` or its variant `clocked async`. The `async` primitive can be used recursively. The body of the `async` executes in parallel with the creating activity, within the context of a `finish` construct.

The `finish` and `clocked finish` primitives act as containers for enclosed activities and wait for their termination before terminating themselves. A `clocked finish` creates an unnamed clock. Each `clocked async` *registers* the created activity to its clock. An activity which terminates is deregistered from the clock. In this way, the number of activities registered to a clock may vary in the course of the execution of a program. A clocked activity may issue an `advance` instruction, in which case it stalls until all presently registered activities also have issued an `advance` instruction. One can show that the use of clocks assigns the activities belonging to the same `clocked finish` to phases which are executed sequentially in order of their *phase number*, the number of `advance` instructions issued by an activity and its ancestors since the initial `clocked finish` [16]. While the present paper targets the X10 language, we believe that the approach applies also to Habanero with minor syntactic modifications.

3 The Basic Scheme

The standard approach for generating a parallel program from a schedule is to construct *fronts*, sets of statement instances that are scheduled at the same time, and to execute fronts sequentially in chronological order (the SEQ of PAR mode in Occam terminology). The clocks of X10 are exactly what is needed for building fronts. A front is the set of statement instances executed between two consecutive `advance` instructions. Since `advance` instructions can be enclosed in multidimensional loops, the phase numbers may be polynomials, and the problem is to adjust these polynomials in such a way that they reproduce the program schedules. This approach is reminiscent of the proposal of using clocks for encoding loop transformations [24].

3.1 X10 Code Generation

Let $(T_i), i \in \{1, \ldots, n\}$ be the list of tasks of the source program, each with its schedule θ_i. Let CP_i be the slice of the control program that encloses task T_i and no other. This program has an iteration vector x_i, an iteration domain D_i, and an execution order which is simply the lexicographic order on x_i, $\ll$. The target program (in X10 notation) has the following shape:

```
clocked finish {
  clocked async {
    for (x_i in D_i) {
      some advance instructions;
      T_i;
    }
    //...
  }
}
```

This program cannot be executed as is. Since there is no way of undoing an `advance` instruction, if we intend that the phase number of iteration x_i be equal to $\theta_i(x_i)$, then θ_i must be a monotonically increasing function with respect to $\ll$. This can be obtained by adjusting the order of the iterations of the x_i loop. According to the official specification of OpenStream, stream pointers are increased at each task creation, hence changing the task creation order may change the pointers values and the semantics of the program. But remember that in the course of dependence calculations, closed form expressions for the stream pointers have been obtained. By a process analogous to induction variable substitution or reverse strength reduction, it is enough to replace each increment by an evaluation of the corresponding form. In this way, any creation order will result in an equivalent program. Note that since the execution order of the main program is entirely dictated by the schedules, and these will not be modified, this transformation will incur no loss of parallelism.

3.1.1 The Problem of the Decreasing Schedule

The property we need to enforce is, for two statement instances u and v in the same activity:

$$u \in D, v \in D, u \ll v \Rightarrow \theta(u) \leq \theta(v), \tag{1}$$

where D is the iteration domain of the activity. A first attempt is to impose (1) as an additional constraint when building θ. However, such a schedule may not exist, as shown by the following OpenStream example:

```
parameter N;
stream s[N+1];    // array of N+1 streams
for(i=0; i<N; i++) {
  task a {                    // theta = N-i
    read once from s[i+1];
    write once into s[i];
  }
}
task b {                      // theta = 0
  write once into s[N];
}
```

The notation `s[i]` refers to the i-th stream in an array of streams `s` (declared as an array of $N+1$ streams).

Task $a(i)$ is created before task $a(i+1)$, but there is a dependence from $i+1$ to i, hence no increasing schedule exists. Changing the creation order: $j = N - i$ solves the problem:

```
parameter N;
stream s[N+1];    // array of N+1 streams
for(j=1; j<=N; j++)
  task a {                    // theta = j;
    read once from s[N-j+1];
    write once into s[N-j];
  }
task b {                      // theta = 0;
  write once into s[N];
}
```

Note that, in this case, there is no need to adjust the evaluation of the stream pointers, which are all zero.

A first step in the solution is to check if (1) is satisfied or not, by testing if the system of constraints

$$\begin{aligned} u \in D \quad &, \quad v \in D, \\ u \quad &\ll \quad v, \\ \theta(u) \quad &> \quad \theta(v) \end{aligned} \tag{2}$$

is feasible or unfeasible. In the latter case, θ is monotonically increasing and no modification of the program is needed. In the same way, by reversing the order of the inequality (2), one can decide if θ is monotonically decreasing, in which case it is enough to replace $\ll$ by its opposite. This is the case for the above example. Finding both systems unfeasible would imply that D is empty, which is forbidden and can easily be checked beforehand. Having both systems feasible implies that one partial derivative of θ at least changes sign in D. Setting this derivative to be positive or negative allows splitting D in two parts which can be processed independently. Experience shows that this situation happens very seldom. As a case in point, the schedule for an infinite loop must ultimately be monotonically increasing.

It remains to explain how to check the feasibility of (2). Since θ may be a polynomial, plain linear programming may not be adequate. The preferred method is to try to show that -1 is a positive linear sum of products of constraints in (2). This approach is similar in spirit to the Fourier-Motzkin algorithm: if one can generate an absurd inequality as a consequence of the system to be checked, this system is unfeasible. The ultimate computation can be delegated to powerful linear programming software, which makes the method very efficient. Other possibilities are submitting the system to the Z3 SMT solver,[2]

[2] `https://github.com/Z3Prover/z3`

(which has been shown to be slower than modern linear programming tools [20]), or using developments in Bernstein polynomials [5].

3.2 How to Generate the "Advance Code"

A very simple solution to the generation of the `advance` instructions and control flow is to assign a local phase counter ϕ to each activity. This counter is set to zero at the beginning of the activity. Before executing each task instance u, insert the following code:

```
n = theta(u);
execute (n-phi) advance instructions;
phi = n;
```

The number of `advance` instructions to be executed can also be computed statically, and, most of the time will be found to be 1. This might be related to the design of bounded delay schedulers.

The following OpenStream example model the conversion of a two-dimensional triangular dataset into a one-dimensional stream:

```
stream s, t;
task reset {
  write once into t;
                // theta = 0
}
for(i=1;; i+=1)
  for(j=0; j<i; j+=1)
    task source {
                // theta = i(i-1)/2 + j
      write once into s;
    }
for(k=0;; k++)
  task sink {    // theta = k+1
    read once from s;
    read once from t;
    write once into t;
  }
```

It is easy to see that this code has monotonically increasing schedules. In the corresponding X10 skeleton the notation `[| source |]` represents the unspecified body of the task `source`:

```
  clocked finish {
    [| reset |]
    clocked async {
      phi1 = 0;
      for(i=1;; i++)
        for(j=0; j<i; j++) {
```

```
      n = i*(i-1)/2 + j;
      for(l=0; l < n - phi1; l++)
        advance;
      [| source |]
      phi1 = n;

  }
}
clocked async {
  phi2 = 0;
  for(k=0;; k++) {
    n = k+1;
    for(l=0; l<n-phi2; l++)
      advance;
    [|sink|]
    phi2 = n;
  }
}
```

The `reset` task has been allocated to the parent activity. Since it is scheduled at time 0, it does not need an `advance` instruction. It terminates immediately, therefore does not interfere with the execution of `source` and `sink`, which proceed in lock step indefinitely. The reader may care to check that all `advance` loops execute only one iteration.

4 Related Work

The construction of polynomial schedules was first considered by Achtziger et al. in [1], but there was no attempt at code generation at that time. Beside the method of [10], polynomial schedules can also be obtained by counting iterations, either in sequential programs [18] or after multidimensional scheduling. Code generation in a polynomial context was first considered in Armin Groesslinger's PhD thesis [13]; here, the focus was on generating code for semi-algebraic iteration domains, either native to the original code or after a loop transformation like tiling. In this sense, this work is complementary to the present proposal. The nearest attempt at polynomial code generation is [4]. The context is quite different: generating code after a non-rectangular loop nest has been transformed into a single loop by linearization. The approach consists in inverting the ranking function to recover the original loop counters, using a standard computer algebra systems, which is limited to the fourth degree. It may be that this limit can be raised by using *cylindrical algebraic decomposition* as in Groesslinger's work. The approach may also be applied to the present problem; a detailed comparison must wait for further work.

Lastly, there has been many attempts, not to handle polynomials, but to avoid them: see for instance *delinearization* algorithms [19] or CRP [9] where

multidimensional streams were proposed to avoid linearization, or in parametric tiling schemes [17, 14, 7] or focus on specific tile aspect ratios [15].

5 Conclusion and Future Work

In this work, we have presented a new solution to the problem of generating parallel code from polynomial schedules. Two observations: first, an affine form is a polynomial of degree one, hence the method can be applied to polyhedral programs. How it compares with standard approaches like CLooG is an open question. Second, observe that each task is processed independently, hence the generation is done in linear time in the number of tasks and is independent of the number of task instances.

The present solution is specific to the selected pair of languages, here OpenStream and X10. We expect that it can be adapted to other situations, like sequential programs, Kahn Process Networks, SDF, Habanero C and Java, and perhaps OpenMP itself.

An interesting question is whether OpenStream and X10 have equivalent expressive power. The answer is probably negative. For instance, OpenStream programs may have deadlocks, while X10 does not, leaving out intricate constructs where clocked and unclocked `async`s are mixed inside the same `clocked finish`. In this sense, the existence of a schedule for the OpenStream program is a certificate of equivalence, and the present proposal is an explicit construction of the equivalent program. Conversely, X10 can have races while OpenStream is single assignment and cannot.

However, the present approach generates only task parallelism, hence the degree of parallelism is bounded by the number of tasks. But an OpenStream program may have data parallelism as for instance when an iterator is missing in a schedule, or when a schedule has rational coefficients. How to generate code in these cases is a subject for future work.

The resulting program may be far from optimal. Consider for instance the following pseudo-OpenStream code:

```
stream s, t;
task a {
  write once in s;    // theta = 0
}
task c {
  read once from t;   // theta = 2;
}
task b {
  read once from s;   // theta = 1;
  write once in t;
}
```

This program is clearly sequential. The generated code seems to be parallel, but its sequentiality is encoded in the number of `advance` instructions preceding each

task. Another case is that of a program having `fork` / `join` parallelism, which can be encoded very simply by generating several `clocked asyncs` instead of only one. A last example is a part of code using only one activity, where `advance` instructions are superfluous. These optimizations can be left to a post-processing pass, perhaps using techniques borrowed from [12].

Lastly, this work is only a slight extension beyond the polyhedral model. How to enlarge its applicability is probably one of the most urgent challenges for the future of compiler construction for parallel languages. Will the solution be in the design of further models, or in approximation methods, or in more powerful mathematical tools, only time will tell.

References

[1] Wolfgang Achtziger and Karl-Heinz Zimmermann. Finding quadratic schedules for affine recurrence equations via nonsmooth optimization. *Journal of VLSI Signal Processing*, 25:235–260, 2000.

[2] Michel Brion. Points entiers dans les polyedres convexes. *Ann. Sci. Ecole Normale Supérieure*, 21(4):653–663, 1988.

[3] Philippe Clauss. Counting solutions to linear and nonlinear constraints through ehrhart polynomials: Applications to analyze and transform scientific programs. In *10th ACM Int. Conf. on Supercomputing, ICS'96*, May 1996.

[4] Philippe Clauss, Ervin Altintas, and Matthieu Kuhn. Automatic collapsing of non-rectangular loops. In *IEEE Int. Parallel and Distributed Processing Symposium (IPDPS)*, pages 778–787, May 2017.

[5] Philippe Clauss and Irina Tchoupaeva. A symbolic approach to bernstein expansion for program analysis and optmization. pages 120–133. Springer, April 2004. LNCS 2985.

[6] Albert Cohen, Alain Darte, and Paul Feautrier. Static analysis of openstream programs. In *6th International Workshop on Polyhedral Compilation Techniques*, 2016.

[7] Alain Darte and Alexandre Isoard. Exact and approximated data-reuse optimizations for tiling with parametric sizes. In *24th Compiler Construction Int. Conf. (CC'15), part of ETAPS'15*, pages 151–170, London, UK, April 2015.

[8] E. Ehrhart. *Polynomes arithmétiques et méthodes des polyèdres en combinatoire*, volume 35 of *International Series of Numerical Mathematics*. Birkauser Verlag, 1977.

[9] Paul Feautrier. Scalable and structured scheduling. *International Journal of Parallel Programming*, 34(5):459–487, May 2006.

[10] Paul Feautrier. The power of polynomials. In *5th International Workshop on Polyhedral Compilation Techniques*, 2018.

[11] Paul Feautrier and Christian Lengauer. The polyhedron model. In David Padua, editor, *Encyclopedia of Parallel Programming*. Springer, 2011.

[12] Paul Feautrier, Eric Violard, and Alain Ketterlin. Improving the performance of x10 programs by clock removal. In *Compiler Construction CC 2014*, number 8409 in LNCS, pages 113–132. Springer, 2014.

[13] Armin Größlinger. *The Challenge of Non-linear Parameters and Variables in Automatic Loop Parallelisation*. PhD thesis, University of Passau, Germany, 2009. `http://nbn-resolving.de/urn:nbn:de:bvb:739-opus-17893`.

[14] Albert Hartono, Muthu M. Baskaran, Cédric Bastoul, Albert Cohen, Sriram Krishnamoorthy, Boyana Norris, J. Ramanujam, and P. Sadayappan. Parametric Multi-level Tiling of Imperfectly NestedLoops. In *Intl. Conf. on Supercomputing (ICS)*, pages 147–157, Yorktown Heights, New York, June 2009.

[15] Guillaume Iooss, Sanjay Rajopadhye, Christophe Alias, and Yun Zou. Cart: Constant aspect ratio tiling. In Sanjay Rajopadhye and Sven Verdoolaege, editors, *Proceedings of the 4th International Workshop on Polyhedral Compilation Techniques*, Vienna, Austria, January 2014.

[16] Alain Ketterlin. Personal communication, 2017. `http://dpt_info-u-strasbg.fr/~alain/hbx/hbx-r48.pdf`.

[17] DaeGon Kim, Lakshminarayanan Renganarayanan, Dave Rostron, Sanjay Rajopadhye, and Michelle Mills Strout. Multi-level tiling: M for the price of one. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, SC '07, pages 51:1–51:12, New York, NY, USA, 2007. ACM.

[18] Vincent Loechner, Benoit Meister, and Philippe Clauss. Precise datalocality optimization of nested loops. *The Journal of Supercomputing*, 21(1):37–76, 2002.

[19] Vadim Maslov. Delinearization: An efficient way to break multiloop dependence equations. *SIGPLAN Not.*, 27(7):152–161, July 1992.

[20] Tony Nowatzki, Michael Sartin-Tarm, Lorenzo De Carli, Karthikeyan Sankaralingam, Cristian Estan, and Behnam Robatmili. A scheduling framework for spatial architectures across multiple constraint-solving theories. *ACM Trans. Program. Lang. Syst.*, 37(1):2:1–2:38, November 2014.

[21] Antoniu Pop and Albert Cohen. Openstream: Expressiveness and Dataflow Compilation of OpenMP Streaming Programs. *ACM TACO*, 9(4):53, 2013.

[22] Vijay Saraswat, Bard Bloom, Igor Peshansky, Olivier Tardieu, and David Grove. X10 language specification version 2.2, March 2012. `x10.sourceforge.net/documentation/languagespec/x10-latest.pdf`.

[23] Sven Verdoolaege. isl: An integer set library for the polyhedral model. In *International Congress on Mathematical Software (ICMS 2010)*, pages 299–302. Springer, 2010. LNCS, 6327.

[24] Tomofumi Yuki. Revisiting loop transformations with X10 clocks. In *Proceedings of the ACM SIGPLAN Workshop on X10*, X10'15, pages 1–6, 2015.