HAL open science

Representing Non-Affine Parallel Algorithms by means of Recursive Polyhedral Equations

Patrice Quinton, Tomofumi Yuki

To cite this version:

Patrice Quinton, Tomofumi Yuki. Representing Non-Affine Parallel Algorithms by means of Recursive Polyhedral Equations. IMPACT 2021 - International Microsystems, Packaging, Assembly and Circuits Technology conference, Hipeac 2021, Jan 2021, Budapest, Hungary. pp.1-11. hal-03518569

HAL Id: hal-03518569

https://hal.science/hal-03518569v1

Submitted on 10 Jan 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Representing Non-Affine Parallel Algorithms by means of Recursive Polyhedral Equations

Patrice Quinton
ENS Rennes
France
patrice.quinton@ens-rennes.fr

Tomofumi Yuki
Inria, Univ Rennes, CNRS, IRISA
France
tomofumi.yuki@inria.fr

Abstract

Polyhedral equations allow parallel program to be expressed, analyzed, and compiled automatically, but they cannot express divide-and-conquer approaches. This limitation is basically due to the affine nature of the dependence functions imposed by the model. In this paper, we describe how this limitation can be overcome by extending a structured polyhedral equational language to recursive calls of polyhedral programs. Doing so, we preserve the affine property inside a given call, whereas the non-affine part is carried by the recursive expression of subsystem calls. We describe the basic mechanisms of this extension, show that the fundamental results of polyhedral equations hold, in particular, the schedule of such a system can be found automatically. We illustrate this approach on several well known algorithms, including the FFT.

Note: this paper is a corrected version of the initial paper that was accepted and presented in the Impact 2021 workshop. The difference between this version and the initial concern section 5.4 and 5.5 uniquely, and are explained in Appendix A.

1 Introduction

The polyhedral model is a framework for representing and transforming computations with regularity that has seen many successes in automatic parallelization. The core of the framework is in the compact representations of families of parameterized program instances as integer polyhedra. The rich set of properties provided by mathematical representations of computations enables many desirable features for automatic parallelization, such as composable transformations and exploration of schedules with ILP.

The powerful representation as polyhedral objects also limits the class of programs to computations that can be expressed as systems of affine recurrence equations [18, 22, 26], or loops with static/affine control [8]. Despite many extensions proposed over the last few decades [2, 3, 11, 12, 21, 25], expanding the applicability of polyhedral techniques is still a topic of interest.

In this paper, we discuss an extension of the polyhedral representation of computations to handle recursive algorithms. Divide-and-conquer algorithms, such as the Fast Fourier Transform, have been known as examples of regular, statically controlled, programs that are not compatible with the polyhedral formalism. The main difficulty is due to non-affine control in recursion—a typical divide-and-conquer takes $logN$ recursive steps.

We show that a structured system of affine recurrences is able to represent such recursive algorithms while preserving the important properties in polyhedral representation. The main challenge is in the scheduling of such structured systems that necessitates the handling of some non-affine terms when computing a complete schedule. We believe that being able to represent such programs within the polyhedral framework is an important step towards exploring algebraic transformations involving recursions, similar to decompositions of linear transforms explored with SPIRAL [10].

In the following, we first illustrate the intuitions of our ideas in Section 2. Then, we introduce the Alpha language and the extensions we use to represent recursive polyhedral programs in Sections 3 and 4. We present a preliminary scheduling algorithm for recursive programs in Section 5 and discuss some of the remaining challenges in Section 6. Finally, we discuss related work in Section 7 and then conclude in Section 8.

2 Intuition with an Example

Fig. 1 displays an Alpha program to compute the minimum value of N numbers. A detailed presentation of Alpha and its syntax is postponed to Section 3, but this program is simple enough to be understood without detailed explanations.

The recurrence equation that does the computation traverses the N numbers and updates a variable X with the current minimum value.

This program is obviously sequential and takes N time units to execute. Actually, if we let $t_X = ai + b$ denote the schedule of X, then it is easy to discover

Impact 2021, January 18–20, 2021, Virtual event
2021.

$$
\begin{array}{l}
\textbf{affine}\ \text{minValue}[N] \rightarrow \{: 1 \le N\} \\
\textbf{in} \\
\quad \text{array} : \{[i] : 1 \le i \le N\}; \\
\textbf{out} \\
\quad \text{minimum} : \{\}; \\
\textbf{local} \\
\quad \text{X} : \{[i] : 0 \le i \le N\}; \\
\textbf{let} \\
\quad \text{X}[i] = \textbf{case}\ \{ \\
\qquad \{: i = 0\} : 0[\,]; \\
\qquad \{: 0 < i\} :\ \textbf{min}\ (\text{X}[i-1], \text{array}[i]); \\
\quad \}; \\
\quad \text{minimum} = \text{X}[N]; \\
.
\end{array}
\tag{1}
$$

Figure 1. Alpha program to compute the minimum of N numbers.

that computing X at time i allows the recurrence to be solved. Then, the schedule of the output $t_{\texttt{minimum}}$ is N.

```
minRec(A[N]) = {
   if (N==1) return A[0];
   left  = minRec(A[0:N/2]);
   right = minRec(A[N/2:N]);
   return min(left, right);
}
```

Figure 2. Recursive computation of minimum

Fig. 2 is an informal description of a recursive program for computing the minimum. It looks for the minimum value of the left and right halves of an array, and returns the smallest one of the two results. There are $O(N)$ instances of the function `minRec` over $logN$ recursive steps to compute the minimum of N values. In this simple example, the only computation performed by an instance of `minRec` is a comparison of the two results. Assuming that this takes unit time, a valid schedule for the `min` operation in all instances of this recursive program is $logN - 1$ (see Fig. 3 for a graphical illustration).

Such a program cannot be expressed in Alpha, and in general, does not fit in the polyhedral model. The purpose of our work is to show how one can extend the model to recursive computations, in such a way that the basic properties of the model still hold. In particular, we concentrate on how to schedule automatically a program, as it is the basis for the analysis and synthesis or parallel implementations.

Scheduling in the polyhedral model consists in finding out an affine function that stamps the time instants of computation of the variables. The schedule depends linearly on the size parameter–here the number of elements N–of the program.

In the recursive program, we are interested in finding a schedule that is legal for a collection of instances that have inter-instance dependences due to recursive calls. In our recursive program, a constant schedule, $t_{\texttt{left}} = t_{\texttt{right}} = 0$; $t_{\texttt{min}} = 1$, is a valid schedule (for an instance) that does not violate intra-instance dependences. In our work, we conceptually separate the schedule into two components, one for intra-instance dependences, and another for placement of instances in the global time while respecting inter-instance dependences. The intra-instance dependences add additional constraints for global scheduling, which is calculated by solving recursive equations leading to the $\log N$ term (for methods to solve recursive equations, see [5] and [4]).

Being able to find an explicit schedule for such a program is interesting for two reasons. The first one is to provide a complexity measure of a parallel implementation of the algorithm, which is always interesting. The second one, less general, is related to the use of an equational polyhedral language to express calculations, either to produce parallel code, or to generate a hardware architecture. In the case of Alpha, both outcomes are considered: the alphaZ approach [27] is meant to produce loop nests amenable to efficient parallel implementation, whereas the MMAlpha approach [19] rewrites the program in order to ultimately obtain an equivalent program that can be translated into a hardware description language such as vhdl.

3 Background: Alpha

In this section we describe the Alpha language used in this paper, which is a slight syntactic variant of the original language [16]. In addition, an extension to the language to represent *while loops* and *indirect accesses*, called alphabets [21], may be used in conjunction with our extensions proposed in this paper.

We use Alpha as a representation of the complete computation and not just the dependences; it may be viewed as glorified recurrence equations with a bit of extra information. The main ideas in this paper do not require the program to be represented in Alpha; reduced dependence graphs could be used, provided some metadata regarding the structure of the program (e.g., subsystem calls) are kept.

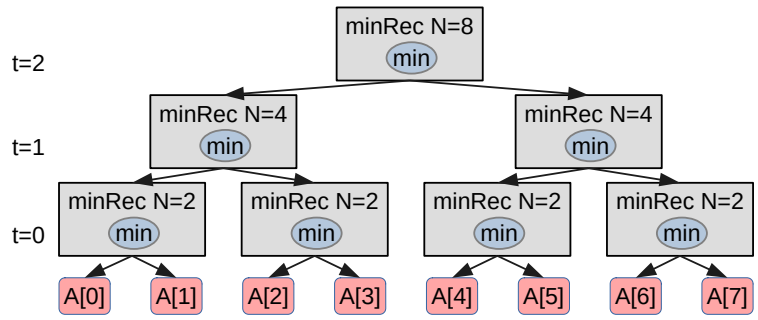

Figure 3. Call structure of recursive min when $N = 8$. The `min` operations performed at each instance of the recursive calls may be given a time stamp as a function of the size parameter, N: $logN - 1$.

3.1 Affine Systems

An Alpha program consists of one or more affine systems that have the following structure:

affine < name >< parameters >
in
(< name > : < domain >)∗
out
(< name > : < domain >)+
let
(< name > = < expr >;)+
.

Each system corresponds to a System of Affine Recurrence Equations (SARE), which consists of the two main components: (i) interface definition, i.e., the domain of inputs and outputs, and (ii) the expressions that define the value of outputs in a purely functional manner. The parameters of a system are given as a parameter domain, which is common for all domains within a system (Parameters represent size parameters of algorithms, such as the sizes N of an array.).

For the sake of simplicity, scalar types of the variables are not described here, and we assume that they are "numbers" with the associated domains being the primary type information. In addition to inputs and outputs, local variables for storing intermediate results may be declared through the keyword **local**.

3.2 Alpha Expressions

The core of an Alpha program is the set of equations that follow the **let** keyword. Such equations have the form `X = e` where `e` is an expression which represents a function from the points of the domain of `X` to some value set–here, "numbers." In general, we would write such an equation

```
X[i,j,…] = e[i,j, …]  ,
```

where indexes `i` and `j` would represents points in the domain of `X`. Parameters are considered as mute indexes of the variables, and are supposed to belong to a polyhedral domain denoted $\mathcal{D}_P$.

In fact, Alpha expressions are built using a few functional operators combining integral functions, as described in Table 1. To each operator corresponds a domain transformation, which assigns a polyhedral domain to the expression from the domains of its constituents. These domains denote where the expression is defined and could be computed.

The semantics of each expression when evaluated at a point z in its domain is defined as follows:

- A constant expression evaluates to the associated constant.
- A variable expression is its value at z, either provided as input or computed through its definition (an equation).
- An operator expression is the result of applying a strict point-wise operator, `op`, on the values of its arguments at z.
- A case expression is the value at z of the branch whose domain contains z. Branches of a case expression are defined over disjoint domains to ensure that the case expression is uniquely defined.
- An if expression `if` E_C `then` E_1 `else` E_2 is the value of E_1 at z if the value of E_C at z is true, and the value of E_2 at z otherwise. E_C must evaluate to a boolean value. Note that the else clause is *required*. In fact, an if-then-else expression in Alpha is just a special (strict) point-wise operator.
- A restriction of E is the value of E at z.
- A dependence expression $f@E$ is the value of E at $f(z)$. The dependence expression in our variant of Alpha uses function joins instead of compositions. For example, $f@g@E$ is the value of E at $g(f(z))$, whereas the original language [16] used $E.g.f$.
- An index expression `val`(f) is the value of f evaluated at point z.

Table 1. ALPHA expressions and their domains. Constants and indexed expressions are defined on the parameter domain. Variables are defined on their declaration domain. The domain of other expressions is obtained using the rules shown in the Expression Domain column.

Expression	Syntax	Expression Domain
Constants	Constant name or symbol	$\mathcal{D}_P$
Variables	V (variable name)	$\mathcal{D}_{\mathsf{V}}$
Operators	$\mathtt{op}(\mathtt{Expr}_1, \ldots, \mathtt{Expr}_M)$	$\bigcap_{i=1}^{M} \mathcal{D}_{\mathtt{Expr}_i}$
Case	$\mathtt{case}\{\mathtt{Expr}_1; \ldots; \mathtt{Expr}_M\}$	$\biguplus_{i=1}^{M} \mathcal{D}_{\mathtt{Expr}_i}$
If	$\mathtt{if}\ \mathtt{Expr}_1\ \mathtt{then}\ \mathtt{Expr}_2\ \mathtt{else}\ \mathtt{Expr}_3$	$\mathcal{D}_{\mathtt{Expr}_1} \cap \mathcal{D}_{\mathtt{Expr}_2} \cap \mathcal{D}_{\mathtt{Expr}_3}$
Restriction	$\mathcal{D}' : \mathtt{Expr}$	$\mathcal{D}' \cap \mathcal{D}_{\mathtt{Expr}}$
Dependence	$f\mathtt{@Expr}$	$f^{-1}(\mathcal{D}_{\mathtt{Expr}})$
Index Expression	$\mathtt{val}(f)$ (range of f must be $\mathbb{Z}^1$)	$\mathcal{D}_P$
Reductions	$\mathtt{reduce}(\oplus, f, \mathtt{Expr})$	$f(\mathcal{D}_{\mathtt{Expr}})$

- $\mathtt{reduce}(\oplus, f, E)$ is the application of $\oplus$ on the values of E at all points in its domain $\mathcal{D}_E$ that map to z by f. Since $\oplus$ is an associative and commutative binary operator, we may choose any order of application of $\oplus$.

Mauras [17] has shown that any expression can be rewritten, thanks to a set of axiomatic transformation rules, in the form `X[z] = f( …)`. This array notation, easier to read than the pure functional one, will be used in the following.

3.3 Context Domain

Each expression is associated with a domain where the expression is defined, but the expression may not need to be evaluated at all points in its domain. Context domain is another expression attribute, denoting the set of points where the expression must be evaluated [6]. The context domain of an expression E is computed from its domain and the context domain of its parent.

The context domain $\mathcal{X}_{\mathsf{E}}$ of the expression E is:

- $\mathcal{D}_{\mathsf{V}} \cap \mathcal{D}_{\mathsf{E}}$ if the parent is an equation for V.
- $f(\mathcal{X}_{E'})$ if E' is $E.f$.
- $f_p^{-1}(\mathcal{X}_{E'}) \cap \mathcal{D}_E$ if E' is $\mathrm{reduce}(\oplus, f_p, E)$.
- $\mathcal{X}_{\mathsf{E}'} \cap \mathcal{D}_{\mathsf{E}'}$ if the parent E' is any other expression.

This distinction of what *must* be computed and what *can* be computed is important when the domain and context domain are used to analyze the computational complexity of a program.

3.4 Synthesis of ALPHA Programs

The standard flow for ALPHA programs [19] is as follows:

- A static semantic analysis checks that the ALPHA program is well-defined. This is essentially a form of type checking verifying that the domain dimensions are compatible and expression domains cover corresponding context domains.
- A schedule for the program is sought. There are various ways of scheduling an ALPHA program, but in this paper, we shall restrict ourselves to unidimensional, affine schedules. Simply speaking, a variable $\mathrm{X}[i]$ in a program of parameter N shall be scheduled at a time $t_X(i, N) = ai + bN + c$.
- Once a schedule is found, a code generator is used to produce an implementation of the computation with a specified schedule for the target platform. The target platform (and hence the generated language) may range from parallel loops for CPUs/GPUs to hardware description languages for FPGAs/ASICs. An important transformation used during this step, especially for HDL generation, is the *change of basis*, which rewrites the program by applying affine transforms to domains of variables.

When extending ALPHA to handle recursive programs, we would like this flow, called synthesis, to be preserved. That is, we need to ensure that the three key components are also extended: (i) expression/context domain calculation, (ii) scheduling, and (iii) change of basis.

4 Language Extensions

In this section, we describe the language extensions to the core ALPHA for supporting recursive programs.

4.1 Subsystem calls

Alpha programs are also structured (modularized) at the granularity of systems. A *subsystem call* is an alternative way to define the values of Alpha variables by invoking instances of another system [7].

A subsystem call is specified as the following:

$$(Y_1, Y_2, ..., Y_m) = < \text{name} > [f](X_1, X_2, ..., X_n)$$

where f is an affine function that maps parameters of the caller to those of the callee. Its semantics are that variables $Y_1, Y_2, ..., Y_m$ are the outputs of the callee system, identified by its name, given $X_1, X_2, ..., X_n$, which are Alpha expressions, as inputs. The original version contains subsystem calls to define variables through a collective invocation of multiple instances of the callee system. We omit this feature in this paper, and assume that a subsystem call equation corresponds to a single invocation of the callee system.

4.2 When Construct

We propose a minor extension to the Alpha language to allow recursive definitions. The main observation is that the language is lacking a clean syntax to specify recursive steps and the base case within a single system. Thus, we extend Alpha by adding a construct to support a top-level `case` expression over the parameters.

This is achieved by adding a **when** construct to the **let** clause in the original syntax, and allowing multiple **let** clauses to exist in a single system:

$$\textbf{when} < \text{domain} > \textbf{let}$$

The different **let** clauses may now specify a different way to compute the outputs depending on the parameter values. How the system interacts with other systems remains unchanged, since the interface (inputs and outputs) is still common among the different **when** clauses.

4.3 Extending Consistency Check

How context domain calculation can be extended to Alpha programs with subsystem calls has been discussed by de Dinechin et al. [7]. The two important steps in extending the context domain calculation are:

- Extend the parameter space to be the product of parameters in both (caller and callee) systems. The two sets of parameters are connected via equalities corresponding to the affine function mapping one set to the other.
- Extend the context domain for input expressions in subsystem calls from the corresponding input variables in the callee.

This extension can be applied as is for recursive Alpha, since the domain calculations only concerns the two directly interacting systems. However, the semantic analysis at the beginning of synthesis needs to ensure that there is no infinite loop. We currently perform a simple test to detect trivial loops, but we believe that existing techniques for termination proof of affine transition systems can be directly used to perform this check [1].

4.4 Influence on Change of Basis

In the typical synthesis explained in Section 3.4 targeting hardware designs, the change of basis (CoB) transformation is used to *reflect* the schedule—one of the dimensions becomes the time after CoB. With recursive programs, this step does not directly apply for two reasons: (i) multiple instances of a system are used for a given computation, and (ii) the schedules are no longer affine.

However, the schedules for recursive systems are split into two components: intra-system and inter-system. Only the inter-system component may be non-affine in our approach. Thus, we may apply CoB for the intra-system component of the schedule as usual. The inter-system component may either be (i) reflected for fixed size parameters (i.e., equivalent to inlining of the subsystems) since the values would then be constants, or (ii) implicitly reflected through recursion, e.g., by module instantiation in case of VHDL.

4.5 `minRec` Example in Alpha

Fig. 4 shows an Alpha recursive program that corresponds to the example discussed in Section 2.

The program has two parts (**when** clauses). The first part holds when the parameter N is equal to 1, and then, the output of the program is just the single element array[1] of the input array.

When N is not 1, the definition is more involved. Variables `array1` and `array2` are meant to contain each one half of the input array, as described by the first two equations after the **let** . Variables `min1` and `min2` are defined recursively by a call to the `minRec` program operating respectively on `array1` and `array2`, and where the parameter N is set to $N/2$.

Although context domain calculation and change of basis carry over to recursive programs almost seamlessly, this is not the case for scheduling. Indeed, we do not know the scheduling of a called system when computing that of the calling system, since the program is recursive.

Moreover, the schedule of a variable X[i] in a system of parameter N will not have the form $t_X(i, N) = ai + bN + c$, otherwise, we would loose the nice divide-and-conquer property of this writing. Instead, we must

```
affine minRec[N] → {: 1 ≤ N}
in
    array : {[i] : 1 ≤ i ≤ N}
out
    minimum : {}
when {: N = 1}
let
    minimum = array[1];

when {: N ≥ 2}
local
    min1 : {}
    min2 : {}
    array1 : {[i] : 1 ≤ i and 2i ≤ N}
    array2 : {[i] : 1 ≤ i and 2i ≤ N}
let
    array1[i] = array[i];
    array2[i] = array[i + N/2];
    (min1) = minRec[N/2](array1);
    (min2) = minRec[N/2](array2);
    minimum = min (min1, min2);
.
```
(2)

Figure 4. Recursive Alpha program for a divide-and-conquer search of the minimum of N numbers.

look for a synthesis that will, explicitly or implicitly, result in a scheduling where the bN part is replaced by a function $\log N$.

To simplify matter, assume that N is a power of 2– we shall see in the following that one can infer easily, at least in such an example, that this constraint is met, during the static semantic analysis.

5 Scheduling Recursive Alpha

In this section, we recall results relative to scheduling affine recurrences (See for example [20].) Scheduling an Alpha program means finding, for each variable V, an integral function of the indexes, say z, of this variable such that for all z, this function is greater than that of expressions of which V depends. Such a function is called a *schedule*. Say that $V(z)$ depends on $W(z')$, and let us denote $t_V(z)$ the schedule of V and $t_W(z')$ that of W. Then, if $V(z)$ depends on $W(z')$ we must have

$$t_V(z) > t_W(z') \quad . \tag{3}$$

Before explaining how schedules can be found, we make two remarks. First, a schedule need not necessary be an integral function–i.e, $t_V(z) \in \mathbb{Z}$. It is sufficient that the schedule values belong to a totally ordered set, and that the values of t are ordered according to the dependences in the Alpha program. Second, the strictness of the condition may be relaxed: actually, what is needed is that no dependence cycle may exist in the Alpha program, and provided this condition is met, dependent expressions may be scheduled at the same time.

For the sake of simplicity, we present our scheduling method in the restricted case when t belongs to $\mathbb{Z}$ and with a strict order condition as shown in (3).

5.1 The Elementary Vertex Method

An Alpha program consists in a set of equations $V = e$, where V is a variable and e an expression. The domain of V, denoted $D(V)$, is a polyhedron of $\mathbb{Z}^n$. Expression e contains occurrences of variables $W(z')$, where z' is an affine expression of z.

Assuming that schedules are affine functions of their indexes, denote $t_V(z) = \tau_V.z+\alpha_Z$ this function[1]. In order for $t_V(z)$ to meet condition of equation (3), it is necessary and sufficient to check this property on the generating system of $D(V)$, i.e., its vertices and rays. In this way, we can replace the potentially infinite number of inequalities by a finite set, leading to solving an ILP.

To illustrate this on simple example, let

$$V(i) = W(i-1) + Z(i-2) \tag{4}$$

be an equation of an Alpha program, and assume that V is defined on the set $D_V = \{i \mid i \geq 0\}$. We then must have

$$t_V(i) > t_W(i-1) \tag{5}$$

$$t_V(i) > t_Z(i-2) \tag{6}$$

for all i, it suffices to show that this property is true on the unique vertex of D_V, i.e., 0, and that the function is not decreasing along the unique ray $r = 1$ of D_V. We are therefore led to the following inequalities

$$\alpha_V - \alpha_W > 0 \tag{7}$$

$$\alpha_V - \alpha_Z > 0 \tag{8}$$

$$\tau_V.1 \geq 0 \quad . \tag{9}$$

With this method, schedules for Alpha programs can be obtained by gathering such inequalities, for all pairs of dependent variables. In practice, the ILP to solve contains a few hundreds of linear equalities, that can be solved in a few tens of a second.

5.2 Alpha Program with Parameters

Interesting Alpha program make use of *size parameters*, as shown in example of Fig. 4, where the parameter N represents the number of elements where the minimum value is sought.

A simple way of finding schedules for parameterized Alpha programs is to make the assumption that the

[1] τ is a vector, and $\tau.z$ represents the dot product of τ and z.

schedule depends on the parameter through an affine relation. Thus, if p represents the vector of parameters, then:

$$t_V(z) = \tau_V.z + \alpha_V + \sigma_V.p \quad . \tag{10}$$

Obviously, this extension does not change in a fundamental way the vertex method shown in 5.1.

5.3 Alpha Programs with Sub-Systems

The above method can be extended to the scheduling of Alpha programs including calls to sub-systems. The simplest way to handle this situation–(See [20])–is to assume that the sub-system has already an affine schedule, and to combine this schedule with that of the calling system.

To be explicit, let

$$(V_1, \ldots, V_k) = \texttt{Callee}[f(p)](W_1, \ldots, W_l) \tag{11}$$

be a call to a subsystem, named here `Callee`, with inputs W_l and outputs V_k (here, we simplify the type of calls, as in general, inputs could be any Alpha expression). The value of the parameters p in the callee system is expressed by an affine function $f(p)$ shown between square brackets.

Since `Callee` has already a schedule, each one of its inputs or outputs has a schedule function t_V or t_W. The schedule depends on the parameter vector, say q, of the called system `Callee`.

Inside the calling system, the schedule of inputs and outputs must correspond to that of the `Callee` subsystem. This, therefore, imposes constraints on the linear part of the schedule of the inputs V or W, τ_V and τ_W. Only the affine constants α_V can be modified to adapt the schedule to the context of the call. Finally, the part σ_V of the scheduling function, related to parameter q, has to be adjusted to the value $f(p)$ of the call.

In equation (11), let for example V be the formal parameter of subsystem `Callee`, corresponding to the actual parameter, say V_1. Let

$$\tau_V.z + \alpha_V + \sigma_V.q \tag{12}$$

be the schedule of V in `Callee`, then the schedule of V_1 must have the form

$$\tau_{V_1}.z + \alpha_{V_1} + \sigma_{V_1}.p \quad . \tag{13}$$

Therefore, $\tau_V = \tau_{V_1}$, $\sigma_V(q) = \sigma_{V_1}.f(p)$.

Using such a method, described informally here, we understand that the vertex method can be extended to subsystem calls: subsystems are scheduled in a bottom-up way, and their schedules are used in calling systems.

Two remarks. First, one may use approaches that do not impose a priori as here, the schedule of subsystems (See [9]). Second, Alpha contains also more elaborated calls to subsystems, called mapped calls; we do not elaborate on this, since recursive calls do not fall under this situation.

5.4 Recursive Alpha

In a recursive Alpha program, we cannot apply directly the method explained in Section 5.3, since the schedule of a called subsystem is not known. However, as we shall see now, scheduling the computations is still possible.

To explain how, we restrict ourselves to the situation that happens in the examples that we consider in this paper, and we assume the following properties:

1. The program contains two **when** clauses, one corresponding to the base part, and the other one to the recursive part.
2. The base part contains no subsystem call, and its parameter domain is reduced to a single point, that we denote p_0.
3. The recursive part contains only plain equations and recursive subsystem calls.
4. In the recursive part, there is no direct dependence path between an input Y of the system and an output X (i.e., such a path always goes through a subsystem call).
5. In the recursive part, there exists no dependence between an output of a subsystem call and the input of another subsystem call.

These properties can be easily checked statically. The analysis of programs that do not meet these properties is beyond the scope of this paper.

In the base part program, the scheduling is found in the same way as in standard Alpha. We thus concentrate on the schedule analysis of variables in the recursive part. We do this by considering successively the latency of a call to the recursive system, then by analyzing its internal delays, and finally, by looking at the schedule of the variables.

5.4.1 Latency of a call. Thanks to properties 4 and 5, the equations of a system can be separated into *prelude* calculations, when they are computed before a recursive subsystem call, and *postlude* calculations, when they are computed after a call to the subsystem.

Let us denote by $\phi(p)$ the latency of a subsystem call with parameter values p, i.e., the maximum number of time instants between its first input and its last output. Then, the latency must be higher or equal to the delay between any input Y and any other output X of the system. However, thanks to properties 4 and 5, any path from Y to X contains a prelude path, a path through a subsystem call, and a postlude path. Let δ be the maximum duration of any prelude path, and γ the maximum

duration of any postlude path, we thus have:

$$\phi(p) \geq \delta + \gamma + \phi(f(p)) \quad . \tag{14}$$

This inequality con be combined with that giving the latency of the base case, which can be computed independently. Let us denote by ϕ_0 the latency of the base case.

5.4.2 Delays. Consider now the values of the δ and of the γ terms. Since schedules are affine uni-dimensional, and since dependencies happen only between variables belonging to the same group (prelude or postlude), the difference between the schedules of two dependent variables is an affine function of the indexes and the parameters, that can be determined by the standard vertex method.

Indeed, consider a dependence between V and W and let $D(V)$ denote the domain of V. Then

$$\max_{z \in D_V} t_V(z) - t_W(z)$$

is obtained by taking the maximum of this expression on the vertices v of D_V. This allows one to obtain the maximum value of δ and of γ, as affines functions of p.

5.4.3 Schedules. Finding out the exact schedule of a variable is not necessary to obtain a valid implementation of the system. Indeed, what is needed is the schedule of computations, relative to the beginning of the call to the subsystem, for the prelude variables, or relative to the end of the call to the subsystem, for the postlude variables. These relative times can be obtained by using the vertex method on the dependencies between prelude variables, or between postlude variables.

5.4.4 Summary. In summary, we can see that finding a schedule amounts to successively:

- Find out the scheduling of the variables in the base case system (using the vertex method).
- Find out the latency ϕ_0 of the base case system.
- Find the scheduling of the prelude variables of the recursive system, relative to the beginning of the call to the subsystem.
- Find the scheduling of the postlude variables of the recursive system, relative to the end of the call to the subsystem.
- Find the maximum delays δ and γ.
- Solve the recursion equations to obtain the latency of a call, as a function of p.

5.5 Scheduling the `minRec` program

Let us apply this method to the program of Fig. 4.

The latency of the base case system is one cycle (needed to read value `array[1]`).

In the recursive part of the program, variables `array`, `array1`, and `array2` are prelude variables. Relative to the beginning of the system evaluation, their schedule is $t_{\text{array}}(i) = 0$, and $t_{\text{array1}}(i) = t_{\text{array2}}(i) = 1$, assuming that there is a copy of the array. Therefore, the value of δ is 1.

Similarly, variables `min1`, `min2`, and `minimum` are postlude variables, and their scheduling, relative to the end of the call to the recursive subsystem is $t_{\text{min1}}(i) = t_{\text{min2}}(i) = 0$, and $t_{\text{minimum}} = 1$, assuming a one cycle delay for the calculation of the minimum operator. Thus, $\gamma = 1$.

The ϕ function can then be obtained by solving

$$\phi(p) = \begin{cases} \phi(\frac{N}{2}) + 2 \text{ if } N > 1 \\ 1 \text{ if } N = 1 \end{cases} \tag{15}$$

which admits the solution $\phi(N) = 2\log_2(N) + 1$. Solutions to recursion equations are

6 Discussion

In this section, we elaborate a little bit on several aspects of this work that require to be explored in more details.

6.1 The FFT

An interesting example is the Fast Fourrier Transform which is well-known not to be represented using standard recurrence equations. Fig. 5 shows a simplified version of the FFT. The input is a vector x of size N, where N is a power of 2. The base part of the system is simply the vector x itself. For $N > 1$, vector x is separated in halves `left` and `right`, on which the same algorithm is called recursively, leading to results `q1` and `q2`. These results are then interleaved and combined using a simple addition – for the sake of simplicity – which should be replaced by the FFT butterfly operation.

The scheduling of this example was done automatically using MMAlpha.

6.2 Static Analysis

To be valid, recursive Alpha programs should be restricted to some values of the parameters. In the examples of `minRec` and `FFT`, the value of N is a power of 2. In general, what we need to check is that the recursion on the parameters lead to values in the base parameter domain. It is easy to check that values of the parameters are decreasing inside a polyhedron, by a simple analysis of the generating system of the parameter domain. Finding out a subset of values in the parameter domain such that the recursion leads to a base case is, in general, more involved and requires to be explored.

```
affine FFT[N] → {: 1 ≤ N}
in
    x : {[i] : 1 ≤ i ≤ N}
out
    y : {[i] : 1 ≤ i ≤ N}
when {: N = 1}
let
    y[i] = x;
.
when {: 2 ≤ N}
local
    left : {[i] : 1 ≤ i and 2i ≤ N}
    right : {[i] : 1 ≤ i and 2i ≤ N}
    q1 : {[i] : 1 ≤ i and 2i ≤ N}
    q2 : {[i] : 1 ≤ i and 2i ≤ N}
    z : {[i] : 1 ≤ i ≤ N}
let
    left[i] = x[−1 + 2 ∗ i];
    right[i] = x[2 ∗ i];
    (q1) = FFT[N/2](left);
    (q2) = FFT[N/2](right);
    z[i] =
      case {
        {: 2i ≤ N} : if i%2 = 0 then
          q1[i] + q1[−1 + i]      else
          q1[i] + q1[1 + i];
        {: N < 2i} : if i%2 = 0 then
          q2[i − N/2] + q2[−1 + i − N/2]   else
          q2[i − N/2] + q2[1 + i − N/2];
      };
    y[i] = z[i];
.
```
(16)

Figure 5. Recursive Alpha program for the FFT.

In many situations, the recursion domain is obvious, as here. Divide-and-conquer strategies with other schemes can be sought (for example, when N is a power of 3).

Note that, in practice, the value of N is fixed – since ultimately, a finite architecture has to be generated –, and therefore, the property can be checked.

6.3 Generating an Architecture

As indicated in 3.4, the generation of a parallel architecture for such programs amounts to rewrite the equations in a new, time-space mapped representation, obtaied using a *change of basis* transformation: the scheduling function, completed by the appropriate spatial mapping, forms a unimodular transformation that can be applied without changing the semantics of the Alpha program. It should be clear that this transformation is valid also on recursive Alpha. In the case of the `minRec` program, we would obtain a tree architecture whose leafs correspond to the base case, and each intermediate level would contain a comparator and a few registers. The `FFT` program would be implemented as $\log_2 N$ levels of butterfly operators. In both cases, a VHDL code could be generated easily, when N is fixed.

6.4 Solving the parameter recurrence

Solving in general recursive equations may not be as easy as here. However, the interested reader will find in [5] or in [4] methods to solve more involved recursive equations. It should also be noticed that obtaining a closed form for the ϕ function may not be necessary: when the value of the parameter is set, and again, this must be the case to generate an architecture, the recursion itself guarantees that the sequence of calls leads to a $\log_2 N$ number of steps.

7 Related Work

The polyhedral model is founded around compact, mathematical, representations of programs as affine recurrence equations [18] and reduced dependence graphs extracted from loop programs [8]. These representations provide rich properties, e.g. closure under affine transformations, as well as important techniques for program optimizations, such as scheduling with ILP, and code generation. Thus, the extension of the polyhedral model to a wider range of programs, while preserving all the important properties, is a continued topic of interest for many years. Benabderrahmane et al. [3] have proposed extensions to handle irregular (data-dependent) control-flows while allowing previously developed optimizations to seamlessly carry over. Similarly, mono-parametric tiling [11, 12] allows a subset of parametrically tiled programs (those with fixed aspect ratio) to be expressed as affine sets. Our work also aims at extending the class of programs amenable to polyhedral techniques by handling recursive programs.

There is also a body of work on extending the applicability of the polyhedral techniques through dynamic compilation [14, 15, 23]. The main idea is that programs that seem non-affine during static analysis may actually have affine behavior when executed. The APOLLO framework [23] uses dynamic profiling to test if irregular memory accesses have (or can be approximated as) affine behavior. Kobeissi et al. analyze programs and detect recurrences that can be replaced by affine loop nests and therefore, can be handled by standard polyhedral techniques [14, 15]. Although our work also concerns recursive programs, our goal is to express parallel

algorithms with recursive structure, rather than analyzing an already recursive program.

Sundararajah and Kulkarni [24] have proposed a framework for analyzing and transforming recursive programs similar to the polyhedral model in that it has the ability to reason and transform at the level of statement instances. However, their representation of recursive programs based on finite-state transducers are not directly compatible with the polyhedral model. Our work is restricted to a much more limited class of recursions, but is an extension to the polyhedral model. The primary target of our work is not arbitrary recursive programs, but polyhedral programs that may benefit from algorithmic optimizations involving recursive decomposition, such as FFT or various sub-cubic algorithms for matrix multiplication.

SPIRAL [10] is a framework for generating efficient implementations of signal processing algorithms, which are expressed as linear transforms. SPIRAL explores decompositions of a linear transform through application of rewriting rules that encode decompositions. Such decomposition is at the core of many algorithmic optimizations (e.g., FFT, Strassen algorithm) that are scarsely explored in polyhedral compilers. We hope that recursive systems provide a way to explore such optimizations for a class of programs that is larger than linear transforms handled by SPIRAL.

Javanmard et al. have proposed a framework for generating efficient divide-and-conquer algorithms for dynamic programming [13]. Their work uses polyhedral techniques to generate portable (mono-parametric [11, 12]) recursive programs. However, they generate recursive programs during code generation, outside of the polyhedral representation. Our work is about representing and analyzing recursive programs represented as a set of affine recurrences.

Initially, subsystems were used as a way to write structured Alpha programs, and subsystems were entirely inlined before scheduling [7]. Quinton later proposed a scheduling technique for Alpha programs with subsystems [20] and Feautrier developed another method for structured scheduling targeting Communicating Regular Processes [9]. These techniques also decouple scheduling of a system and the scheduling across systems, but the main motivation is scalability. Our work is different in that we compute a schedule for multiple instances of the same system that is recursively called.

8 Conclusion

We have shown that recursive systems of polyhedral equations can represent divide-and-conquer approaches to parallel algorithms, and we have explained how a language such as Alpha can be extended to handle such recursions, by using sub-systems calls together with conditions on the values of the parameters. We have explained how the scheduling of such programs can be obtained automatically using the methods which handle classical recurrence equations.

We have illustrated this extension on the FFT algorithm, and explained that all the properties needed to generate hardware for such kinds of algorithms can be naturally and simply extended.

We already have presented several research avenues for this work: multi-dimensional scheduling, extension of the axiomatic transformations such as normalization, change of basis, etc. This requires a detailed semantics of the extended language to be described, in order to certify the validity of these transformations.

Implementing a high-level synthesis tool for recursive Alpha is also an interesting research goal, since it would increase the power of MMAlpha or alphaZ tools.

Finally, combining recursivity and reduction operators would provide a strong basis for high-level transformation of algorithms, aiming at targeting various architectures, as is required by practical applications of parallelism.

Acknowledgments: the authors would like to thank the referees for their corrections and suggestions. We also thank Sven Verdoolaege who carefully read this paper and helped correct several errors.

A Corrections made to the first version

After this paper was accepted, the attention of the authors was drawn to an error that appeared to be quite significant, and therefore, the authors made changes to the paper in order to correct it. Corrections concern Sections 5.4 and 5.5 about scheduling. In the initial version, it was assumed that the schedule of a variable in a recursive call would have the form $t_V(z) = \tau_V z + \alpha_V + \phi(p)$ where ϕ would be a function determined by a recursion equation leading to a logarithmic term in the parameter p. Actually, as is now presented in this version, the logarithmic term represents the *latency* of a system call, and it is also obtained by solving recursion equations, but it cannot be used directly as an offset in any variable schedule. We have to separate the variables between those that are computed before a recursive call, from those that are computed after it, as is done here. In order to make this presentation easy to understand, we have restricted ourselves to a simple form of divide-and-conquer algorithm that is characterized in Section 5.4.

References

[1] Christophe Alias, Alain Darte, Paul Feautrier, and Laure Gonnord. 2010. Multi-dimensional Rankings, Program Termination, and Complexity Bounds of Flowchart Programs. In *Static Analysis*, Radhia Cousot and Matthieu Martel (Eds.). 117–133.

[2] Denis Barthou, Jean-François Collard, and Paul Feautrier. 1997. Fuzzy Array Dataflow Analysis. *J. Parallel and Distrib. Comput.* 40, 2 (1997), 210 – 226. https://doi.org/10.1006/jpdc.1996.1261

[3] Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul. 2010. The Polyhedral Model is More Widely Applicable than You Think. In *Proceedings of the 19th Joint European Conference on Theory and Practice of Software, International Conference on Compiler Construction (CC '10)*. 283–303. https://doi.org/10.1007/978-3-642-11970-5_16

[4] Anne Benoit, Yves Robert, and Frédéric Vivien. 2013. *A Guide to Algorithm Design: Paradigms, Methods, and Complexity Analysis*. Chapman & Hall/CRC. 380 pages. https://hal.inria.fr/hal-00908448

[5] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2001. *Introduction to Algorithms* (2nd ed.). The MIT Press.

[6] F. De Dinechin. 1997. *Structured systems of affine recurrence equations and their applications*. Technical Report. IRISA-PI–97-1151, Publication interne IRISA.

[7] F. de Dinechin, P. Quinton, and T. Risset. 1995. Structuration of the ALPHA language. In *Programming Models for Massively Parallel Computers*. 18–24. https://doi.org/10.1109/PMMPC.1995.504337

[8] Paul Feautrier. 1991. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming* 20, 1 (1991), 23–53. https://doi.org/10.1007/BF01407931

[9] Paul Feautrier. 2006. Scalable and Structured Scheduling. *International Journal of Parallel Programming* 34, 5 (2006), 459–487.

[10] F. Franchetti, T. M. Low, D. T. Popovici, R. M. Veras, D. G. Spampinato, J. R. Johnson, M. Püschel, J. C. Hoe, and J. M. F. Moura. 2018. SPIRAL: Extreme Performance Portability. *Proc. IEEE* 106, 11 (2018), 1935–1968. https://doi.org/10.1109/JPROC.2018.2873289

[11] Guillaume Iooss. 2016. *Detection of linear algebra operations in polyhedral programs*. Ph.D. Dissertation. Colorado State University and ENS Lyon.

[12] Guillaume Iooss, Sanjay Rajopadhye, Christophe Alias, and Yun Zou. 2014. Constant Aspect Ratio Tiling. In *Proceedings of the 4th International Workshop on Polyhedral Compilation Techniques*, Sanjay Rajopadhye and Sven Verdoolaege (Eds.). Vienna, Austria.

[13] Mohammad Mahdi Javanmard, Zafar Ahmad, Martin Kong, Louis-Noël Pouchet, Rezaul Chowdhury, and Robert Harrison. 2020. Deriving Parametric Multi-Way Recursive Divide-and-Conquer Dynamic Programming Algorithms Using Polyhedral Compilers *(CGO '20)*. 317–329. https://doi.org/10.1145/3368826.3377916

[14] Salwa Kobeissi and Philippe Clauss. 2019. The Polyhedral Model Beyond Loops - Recursion Optimization and Parallelization Through Polyhedral Modeling. In *IMPACT 2019 - 9th International Workshop on Polyhedral Compilation Techniques, In conjunction with HiPEAC 2019*. Valencia, Spain. https://hal.inria.fr/hal-02059558

[15] Salwa Kobeissi, Alain Ketterlin, and Philippe Clauss. 2020. Rec2Poly: Converting Recursions to Polyhedral Optimized Loops Using an Inspector-Executor Strategy. In *SAMOS 2020: Embedded Computer Systems: Architectures, Modeling, and Simulation*. 96–109. https://doi.org/10.1007/978-3-030-60939-9_7

[16] H. Le Verge, C. Mauras, and P. Quinton. 1991. The ALPHA language and its use for the design of systolic arrays. *The Journal of VLSI Signal Processing* 3, 3 (1991), 173–182.

[17] C. Mauras. 1989. Alpha : un langage équationnel pour la conception et la programmation d'architectures parallèles synchrones. Thèse de l'Université de Rennes 1, IFSIC.

[18] Patrice Quinton. 1984. Automatic Synthesis of Systolic Arrays from Uniform Recurrent Equations. In *Proceedings of the 11th Annual International Symposium on Computer Architecture (ISCA '84)*. 208–214. https://doi.org/10.1145/800015.808184

[19] P. Quinton, A. Chana, and S. Derrien. 2012. Efficient hardware implementation of data-flow parallel embedded systems. In *2012 International Conference on Embedded Computer Systems (SAMOS)*. 364–371. https://doi.org/10.1109/SAMOS.2012.6404202

[20] Patrice Quinton and Tanguy Risset. 2001. Structured Scheduling of Recurrence Equations: Theory and Practice. In *Proceedings of the Embedded Processor Design Challenges: Systems, Architectures, Modeling, and Simulation (SAMOS '01)*. 112–134. https://doi.org/10.1007/3-540-45874-3_7

[21] S. Rajopadhye, G. Gupta, and DG. Kim. 2011. Alphabets: An Extended Polyhedral Equational Language. In *Proceedings of the 13th Workshop on Advances in Parallel and Distributed Computational Models*, Fujiwara Nakano, Bordim (Ed.).

[22] Sanjay V Rajopadhye and Richard M Fujimoto. 1990. Synthesizing systolic arrays from recurrence equations. *Parallel Comput.* 14, 2 (1990), 163 – 189. https://doi.org/10.1016/0167-8191(90)90105-I

[23] Aravind Sukumaran-Rajam and Philippe Clauss. 2015. The Polyhedral Model of Nonlinear Loops. *ACM Trans. Archit. Code Optim.* 12, 4, Article 48 (Dec. 2015), 27 pages. https://doi.org/10.1145/2838734

[24] Kirshanthan Sundararajah and Milind Kulkarni. 2019. Composable, Sound Transformations of Nested Recursion and Loops. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19)*. 902–917. https://doi.org/10.1145/3314221.3314592

[25] Sven Verdoolaege, Hristo Nikolov, and Todor Stefanov. 2013. On Demand Parametric Array Dataflow Analysis. In *Proceedings of the 3rd International Workshop on Polyhedral Compilation Techniques*, Armin Größlinger and Louis-Noël Pouchet (Eds.). 23–36.

[26] Y. Yaacoby and P. Cappello. 1988. Scheduling a system of affine recurrence equations onto a systolic array. In *Proceedings. International Conference on Systolic Arrays*. 373–382. https://doi.org/10.1109/ARRAYS.1988.18077

[27] Tomofumi Yuki, Gautam Gupta, DaeGon Kim, Tanveer Pathan, and Sanjay Rajopadhye. 2012. AlphaZ: A System for Design Space Exploration in the Polyhedral Model. In *Proceedings of the 25th International Workshop on Languages and Compilers for Parallel Computing* (Tokyo, Japan) *(LCPC '12)*. 17–31. https://doi.org/10.1007/978-3-642-37658-0_2