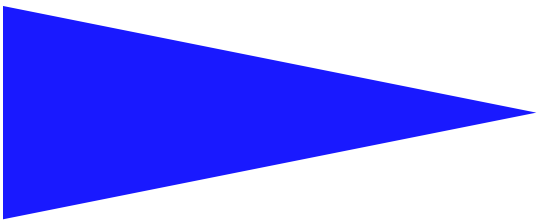
I R I S A
INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES

PUBLICATION
INTERNE
N° 828



USING STATIC ANALYSIS TO DERIVE IMPERATIVE CODE FROM ALPHA

PATRICE QUINTON, SANJAY RAJOPADHYE, DORAN WILDE

IRISA
CAMPUS UNIVERSITAIRE DE BEAULIEU - 35042 RENNES CEDEX - FRANCE

IRISA
INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES
Campus de Beaulieu – 35042 Rennes Cedex – France
Tél. : (33) 99 84 71 00 – Fax : (33) 99 84 71 71

Using Static Analysis to Derive Imperative Code from ALPHA*

Patrice Quinton,** Sanjay Rajopadhye,*** Doran Wilde****

Programme 1 — Architectures parallèles, bases de données, réseaux
et systèmes distribués
Projet API

Publication interne n ° 828 — May 1994 — 18 pages

Abstract: In this article, we demonstrate a translation methodology which transforms a high level algorithmic specification written in the ALPHA language to an imperative data parallel language. ALPHA is a functional language which was designed to facilitate the kinds of static analyses needed for doing regular array synthesis. We show that the same methods which are used for solving regular array synthesis problems can be applied to the compilation of ALPHA as a functional language.

We informally introduce the ALPHA language with the aid of examples and explain how it is adapted to doing static analysis and transformation. We first show how an ALPHA program can be naively implemented by viewing it as a set of monolithic arrays and their filling functions, implemented using applicative caching. We then show how static analysis can be used to improve the efficiency of this naive implementation by orders of magnitude. We present a compilation method which makes incremental transformations on the abstract syntax tree of an ALPHA program in order to make performance improvements and optimize it for a given architecture. The compilation steps described include scheduling, alignment, partitioning, allocation, loop nest generation, and code generation and they are illustrated with a running example. We discuss some of the static analysis issues which come up at each of these steps and briefly describe what static analysis tools are used.

Key-words: parallelizing compilers, affine recurrence equations, polyhedra, linear programming, affine dependencies, dependency analysis, static control programs, monolithic arrays, applicative caching.

(Résumé : tsvp)

*Supported by NSF grant MIP-910852 and Esprit Basic Research Action NANA2, Number 6632
**email: quinton@irisa.fr
***email: rajopadh@irisa.fr
****email: wilde@irisa.fr

Centre National de la Recherche Scientifique (URA 227) Université de Rennes 1 – Insa de Rennes
Institut National de Recherche en Informatique et en Automatique – unité de recherche de Rennes

L'application de l'analyses statique à la derivation de code d'une programme fonctionelle

Résumé : Ce papier présente une méthodologie de transformation d'une spécification algorithmique de haut niveau codée en ALPHA en un langage impératif à parallélisme de données. ALPHA est un langage fonctionnel dédié aux types d'analyse statique requis pour la synthèse de réseaux réguliers. Nous montrons que les méthodes de résolution des problèmes liés à la synthèse de réseaux réguliers peuvent être appliquées à la compilation de ALPHA en tant que langage fonctionnel.

Pour cela, après avoir introduit de manière informelle le langage ALPHA, son adaptation à l'analyse statique et aux transformations de programmes est explicitée. Ensuite, nous présentons une implémentation naïve d'un programme ALPHA sous forme d'un ensemble de tableaux monolithiques dont les fonctions de remplissage font appel à la technique de cache applicative. Grâce à l'analyse statique, l'efficacité de cette première implémentation est améliorée de plusieurs ordres de grandeur. Enfin, nous présentons une méthode de compilation qui utilise des transformations successives sur l'arbre syntaxique abstrait du programme ALPHA en vue d'en améliorer les performances et de l'optimiser pour une architecture cible donnée. Les étapes de cette compilation sont constituées par l'ordonnancement, l'alignement, le partitionnement, l'allocation, la génération de nids de boucles et la génération de code. Elles sont illustrées à travers le déroulement d'un exemple. Pour chaque étape, les problèmes posés par l'analyse statique sont discutés et, une brève présentation des outils utilisés pour les solutionnés est donnée.

Mots-clé : parallélisation de compilateurs, équations récurrentes affines, polyhèdres, dépendances affines, analyse des dépendances, programmes de contrôle statique, tableaux monolithique, cache applicative

1 Introduction

It has long been known that functional programming languages are inherently parallel [35, 3]. However the translation of a functional program to an efficient parallel imperative program has proven to be a very difficult problem [34]. In this paper, we demonstrate a translation methodology which transforms a high level algorithmic specification written in ALPHA, a functional language, to an imperative parallel language.

ALPHA is a single assignment, equational language based on the formalism of systems of affine recurrence equations which have proved useful in the design of regular array architectures. Being a functional language, ALPHA implicitly supports concurrency and communication. Algorithms can be represented at a very high level in ALPHA, as a set of parallel mutually recursive equations, close to how one might specify them mathematically. ALPHA programs tend to be short, intuitive, and easy to verify, although their naive execution is inefficient in both speed and the use of memory. The strength of the ALPHA language lies in the fact that any ALPHA program can be analyzed statically in order to deduce certain information useful for doing optimization. This paper will demonstrate that based on this static analysis, an ALPHA program can be optimized to execute on a target parallel machine by selecting appropriate transformations.

ALPHA was originally developed as a specification language for synthesizing regular arrays from affine recurrence equations. In order to find timing and allocation functions, or map registers and communication links to hardware, all domains and dependencies must first be statically known. Then, the above problems can be cast in terms of doing operations on polyhedra and solving linear programming problems. ALPHA was designed to facilitate these kinds of analyses. Now we are investigating the the simulation of ALPHA programs, and we are encountering the same issues that are found in the compilation of arrays in function languages. We find that the same methods which are used for solving regular array synthesis problems can be used for the compilation of ALPHA as a functional language.

In this paper, we discuss some of the static analysis issues which come up during the development of an ALPHA compiler. To illustrate these issues, we present an example in which we translate an ALPHA program into an imperative data parallel program [13]. This target was chosen because recurrence equations are data parallel by nature. We use a transformational approach to derive an imperative data parallel program which is equivalent by construction to the source program. The transformations are specified according to the results of doing static analysis and information about the target architecture, in order to optimize the program for a specific machine. This methodology separates the issues of algorithm design from the management of architecture dependent parallelism and communication. It allows the programmer to focus on designing the algorithm while delegating the management of parallelism and communication to the compiler.

The paper is organized as follows. In section 2, we present an informal introduction to the ALPHA language and introduce the example. In section 3, we describe the compilation of ALPHA and in section 4, we illustrate it with the aid of an example. In section 5, we

describe briefly what static analysis tools were used. In section 6, we make observations about the compilation of function languages and summarize the work.

2 Introduction to the ALPHA Language

ALPHA was originally designed in the context of systolic array synthesis [18, 21]. It is a pure functional language and as such, is referentially transparent. ALPHA is similar to Crystal [6, 42], another functional language. However, ALPHA is more restricted than Crystal and is based on denotational semantics which allow the index domain of any ALPHA expression to be statically computed. The restrictions in ALPHA were designed into the language to guarantee closure on a set of important program transformations, and also ensure that any ALPHA program may be put into a predefined *normal form*. In most functional languages, linear lists are the dominant data structures used in a program. In ALPHA, there are no lists and no streams. Instead, all variables are type declared at the beginning of a program as polyhedral data fields.

```
system Fibonacci ()
       returns ( fib  : { i | 0<=i } of integer );
let
  fib[i] = case
             {i | i<2  } : 0[];
             {i | i>=2 } : fib[i-1] + fib[i-2];
           esac;
tel;
```

Example 1 : Fibonacci

```
system ForwardSubstitution ( parameter : { N | N>0 };
                             A : { i,j | 0< i<N; 0<=j<i } of integer;
                             B : { i   | 0<=i<N } of integer )
                   returns ( X : { i   | 0<=i<N } of integer );
var  b : { i,j | 0<=i<N; 0<=j<i } of integer;
let
  b[i,j] = case
             {i,j | j = 0 } : A[i,j] * X[j];
             {i,j | j > 0 } : b[i,j-1] + A[i,j] * X[j];
           esac;
  X[i] =   case
             {i   | i = 0 } : B[i];
             {i   | i > 0 } : B[i] - b[i,i-1];
           esac;
tel;
```

Example 2 : Forward Substitution

Figure 1: Two ALPHA programs

In this section, we informally introduce the ALPHA language with the aid of the two examples in figure 1. Example 1 is a system of recurrences for the Fibonacci series and example 2 is a system of recurrences which solve a lower triangular (with unit diagonal)

system of linear equations by forward substitution. An ALPHA program is a `system` of recurrence equations. A `system` declaration consists of a name with a set of input variable declarations and a set of output variable declarations. For instance:

```
system ForwardSubstitution ( parameter : { N | N>0 };
                             A : { i,j | 0<i<N; 0<=j<i } of integer;
                             B : { i | 0<=i<N } of integer )
                   returns ( X : { i | 0<=i<N } of integer );
```

A fundamental feature of ALPHA, which sets it apart from other functional languages, is that all variables in ALPHA are strongly typed and based on polyhedral index domains. A variable denotes a mapping from an *index domain* (the set of all integral points within a specified polyhedron) to values in the *value domain* (integers, reals, or booleans). A polyhedron is bounded by a finite number of linear inequalities, $\{z \in Z^n | Az \geq b\}$. The syntax of a polyhedron in ALPHA is (for example):

```
{ i,j | 0<i<N; 0<=j<i }
```

and the syntax of the declaration of an integer variable A based on that index domain is:

```
A : { i,j | 0<i<N; 0<=j<i } of integer;
```

A system may also have local variables, which are declared after the system header using the keyword `var`. The system of equations that define the variables follow the declarations and are delimited by the pair of keywords `let` and `tel`. The language is equational, and each equation $\texttt{variable}_{LHS}\texttt{= expression}_{RHS}$ names a variable on the LHS and has an expression on the RHS. The denotational semantics of ALPHA [21], specify that, like variables, every ALPHA expression also denotes a function from indices to values. These semantics are fully compositional, and form the basis of a transformational system.

Another important concept used in ALPHA is the notion of a *dependency*, which is an affine function of indices relating the index domain of the LHS variable to the index domain of a RHS variable. Syntactically, a dependency function is written as:

```
(index, index, ... → index-expr, index-expr, ...)
```

where each `index-expr` is an affine expression of the indices to the left of the arrow and of system parameters. Examples of affine dependency functions are `(i->i-2)` and `(Z,i,k->Z-1,2Z-k-1,k-1)`. An example of an equation which uses affine functions to relate domains on the RHS to the domain on the LHS is written explicitly, using dependence functions as: `X = B.(i->i) - b.(i->i,i-1);` or equivalently, it could also be written implicitly, using the so–called *array notation* of ALPHA : `X[i] = B[i] - b[i,i-1];`

The syntax for ALPHA an expression is presented below[1].

[1] Vertical bar is an alternate, square bracket is an optional construct and the asterisk is a Kleene star.

`data-variable | constant`	
`[expression] op expression`	`op` is a binary or unary operator
`expr . dep`	an expression composed with an affine dependency function
`domain : expr`	an expression restricted to a particular (sub)domain
`case expr* esac`	a union of expressions defined over disjoint subdomains

There is also a strict *If* operator which is simply considered as a 3-ary `op` and a reduction operator which applies an associative and commutative binary operator and a many to one index mapping function to a variable [17].

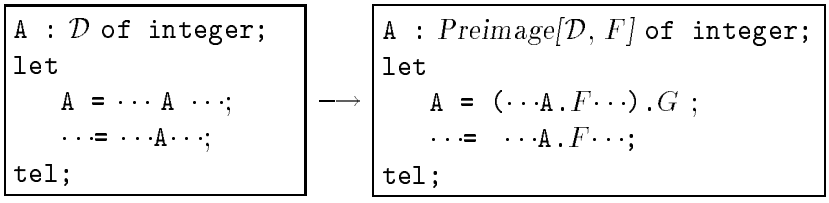
As mentioned above, ALPHA restricts the kinds of programs which can be represented to those where the index domains of variables are unions of convex polyhedra, and data dependencies are affine index functions. These restrictions limit the expressive power of ALPHA, though not to the point of rendering it trivial. A proper subset of the language (uniform ALPHA) is Turing complete, and there are analysis issues about an ALPHA program that are undecidable [31]. A great number of linear algebra and digital signal processing algorithms fall within these limitations. In return for this reduced expressive power, we can make use of a set of powerful static analysis tools that enable us to deduce optimizations to dramatically improve the performance of a program.

2.1 ALPHA Transformations

2.1.1 Change of Basis

Of all the transformations which can be done on an ALPHA program, the *change of basis* is the most common. It is similar to the reindexing of loop variables [40] and other loop transformations done in vectorizing and parallelizing compilers. Given a valid program and a change of basis of a variable, a new, provably equivalent program is derived which not only includes the change of basis of the variable, but also the reindexing of all array references to that variable.

A change of basis of variable `A` using the affine transformation function F can be performed if there exists another affine function G, such that[2] $\forall y \in \mathcal{D} : (F \circ G)(y) = y$, and is done by syntactically rewriting an ALPHA program as follows:

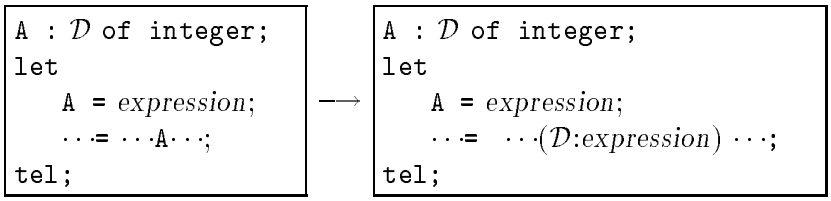


2.1.2 Substitution

An immediate consequence of referential transparency it that equality is substitutive — equal expressions are always and everywhere interchangeable. An equation specifies an

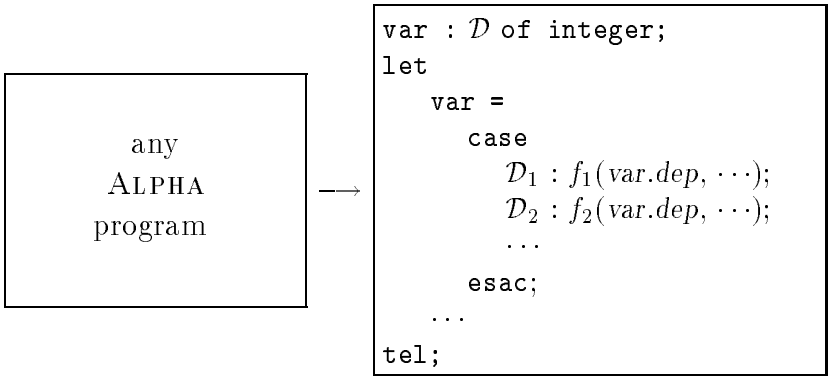
[2] Clearly, if F is invertible, then $G = F^{-1}$ and $(F \circ G)(y) = y$ for all y. This is unnecessarily restrictive. All that is really needed is for F to have a *right inverse* for the points in the domain of F.

assertion on a variable which must always be true. Reasoning about ALPHA programs can thus be done in the context of the program itself, and relies essentially on the fact that ALPHA programs respect the *substitution principle*. Thus, any instance of a variable on the left hand side of any equation may be replaced with the right hand side of its definition. This transformation is done by rewriting an ALPHA program as follows:



2.1.3 Normalization

There is also an ALPHA transformation that "normalizes" any ALPHA program into a predefined syntactic form which fixes the nesting order of ALPHA constructs to : (going from outermost to innermost) case statements, restriction operations, binary and unary operations, and finally dependency functions, variables and constants. This transformation consists of about 35 rewrite rules which among other things change the nesting order of incorrectly nested operations, flatten nested cases, and combine restriction and affine functions where possible. The rewrite rules are confluent and guaranteed to terminate at a stable fix–point [21]. This transformation is often used as a follow up to other transformations (such as the change of basis and substitution transformations presented above) and is where the computation of new domains and dependency functions is done.



3 The ALPHA Compiler

The ALPHA language and transformation system has proved very useful for the synthesis of systolic arrays. In order to enable a user to debug the initial specification and to make the system more useful, we investigated making ALPHA executable, and are working on a compiler. We have encountered many issues that also arise in general compilation, particularly in the efficient implementation of arrays in functional languages. Because of the simplicity of the language, many of these issues can be solved elegantly, using the polyhedral library [38] and linear programming.

In functional languages, arrays are treated as either *incremental* or *monolithic* [36]. Programmers may view incremental arrays as a conventional (and hence, persistent) data structure, manipulated explicitly with an `update` operation, whose semantics denote a function that returns a new array, thereby retaining referential transparency. If implemented naively, incremental arrays require unacceptably high amount of copying, and there has been a body of work to ameliorate this [14, 33, 32]. On the other hand, monolithic arrays [36] consider an array to be simply a map from indices to values. This leads to an elegant programming style which remains declarative and is easy to use. Given that every expression in ALPHA denotes a function from a polyhedral domain to a value, it is natural to consider a program as a collection of monolithic arrays, and its main body as a specification of the "filling" function.

A standard approach to efficient implementation of recursive programs is tabulation [4, 11], also called applicative caching [16]. Here, recomputation is avoided by maintaining a table (or cache) of previously computed values. This makes the underlying implementation imperative, but preserves a purely applicative interface to the user (i.e., referential transparency is retained). This technique is particularly attractive when all the possible arguments to the function, are known statically. It is thus, an obvious method for implementing monolithic arrays: one simply allocates an empty table of the appropriate size, and fills it as needed. This enables the compiler to generate code (albeit naive) when no static analysis information is available (or when the analysis yields a negative result). The code first allocates storage (arrays or tables) for the *entire* domains of all finite variables, and then fills up this storage by computing the values, using demand-driven evaluation. This involves the following steps:

- Visit each integer point in the domain of each output variable. This is a classic problem called *scanning* a polyhedron. A loop program that does this can be generated using the well known Fourier-Motzkin elimination and has been implemented in the polyhedral library of the ALPHA system [1, 19] (the order of scanning being the lexicographic order). At each point, the value of the variable at that index point is computed using demand driven evaluation:
  - Get the arguments: i.e., read the values in the corresponding tables. If these values are not yet known (indicated by a tag bit) we suspend and first evaluate the argument.
  - Apply the filling function, i.e., the RHS of the ALPHA equation. Store the result in the table (with tag set) and also return the value.

This naive code suffers from two drawbacks, namely the overhead of context switching for the demand driven evaluation, and memory inefficiency. Also, if the program has infinite domains, we will need dynamic memory management, and may not be able to completely avoid reevaluation. The compilation can be viewed as a transformation process to incrementally improve this naive code. The following analysis questions are posed:

- Can a *schedule* be *statically* determined? If so, the domains can be scanned in an order consistent with this schedule, without any context switching. This problem is a generalization of the scheduling problem in systolic synthesis [29, 27] to multidimensional schedules [9, 10], and can be cast in terms of linear programming and polyhedral problems.

- An immediate consequence of scheduling analysis is that if a schedule exists, we can immediately detect any parallelism (the dimension of the schedule is strictly smaller than the dimension of the system). In such a case, we may partition the domains of different variables and assign each partition to a different processor. A static analysis of the communication is needed to choose the best partitioning. This can also be cast into the framework of linear programming and polyhedra [28].

- Regardless of whether parallelism is possible or not, the code that is generated so far is memory inefficient, since each table entry is written exactly once. Knowing a schedule, one can perform static lifetime analysis to determine whether and when the memory can be reused. Once again, this analysis can be cast in terms of (linear and nonlinear) optimization problems using the polyhedral domains and affine dependencies of ALPHA. This is a generalization of the *linear allocation functions* used in systolic array synthesis.

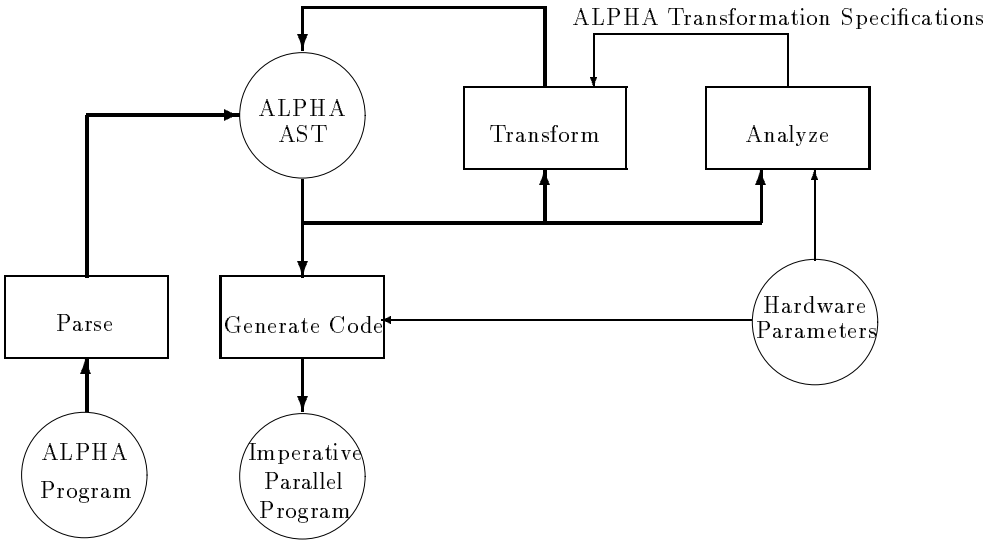


Figure 2: Process Flow for Compiling ALPHA

Based on the above discussion, we are developing a transformational compiler for ALPHA. To simplify the final mapping to imperative parallel code (code generation), an ALPHA program is first transformed into a form which is close to the target language. This is possible since ALPHA has the expressive power to represent a program at many levels. Program transformations are facilitated by the ALPHA language environment, which has

a rich set of proven transformations which can be performed on a program, such as the change of basis, substitution, and normalization transformations described above. When these transformations are thoughtfully and artfully combined, the ALPHA environment becomes a synthesis tool. Figure 2 is a simple diagram of the two-stage synthesis process of ALPHA. The analysis—transformation synthesis loop is followed by the code generation phase. This is the model we use to compile an ALPHA program.

The compilation method shown here operates at a high level of abstraction by performing transformations on the abstract syntax tree (AST) of an ALPHA program. The transformational compiler is based on the following steps which have been introduced above, and which are illustrated in the next section.

1. Parse ALPHA into an internal representation
2. Independent scheduling of variables and computation
3. Alignment of variables
4. Partitioning and Allocation of variables and communication
5. Loop Nest Generation
6. Code Generation

4 Compiling the Forward Substitution Example

In this section, we illustrate the above compilation strategy using the forward substitution program (Example 2, Figure 1).

During the first step, ALPHA is read and parsed, producing an AST which is read into a symbolic algebra system where it is analyzed and transformed symbolically. The static analysis of the AST is simplified by employing the dependency table, which is extracted from the AST and which enumerates the flow dependencies between variables. The dependency table holds the same information as the data flow graph used in [37]. The dependency table for this example is shown below. (The notation $\mathcal{D} : A \rightarrow B$ means that over the index domain $\mathcal{D}$, the computation of A depends on B).

```
{i,j | i-j-1>=0;j>=0;-i+N-1>=0} : b[i,j] -> b[i,j-1]
{i,j | i-j-1>=0;j>=0;-i+N-1>=0} : b[i,j] -> A[i,j]
{i,j | i-j-1>=0;j>=0;-i+N-1>=0} : b[i,j] -> X[j]
{i   |-i+N-1>=0;i>=0          } : X[i]   -> B[i]
{i   |-i+N-1>=0;i>=0          } : X[i]   -> b[i,i-1]
```

In the following subsection, we show how an ALPHA program can be naively implemented. The transformations presented in subsequent subsections can thus be thought of as optimizations on this naive code.

4.1 Naive code generation

We first describe a naive implementation which is guaranteed to be correct, regardless of efficiency concerns. We allocate a table for each variable in the ALPHA program, the size of which is determined by the variable's domain. We traverse all the points in the domain(s) of the output variable(s) using any standard technique [1]. At each point, we

compute the value using a demand driven strategy. If a domain is infinite, we only allocate storage for a finite convex subset which is derived from the original domain by some simple operation such as projection, or "slicing" between two hyperplanes. When a value outside this subdomain is needed, one has to make room for it by discarding a part of the table and this may imply recomputation. The issues such as replacement policies, etc, are similar to those studied in classic tabulation [4], and all those results can be used. Clearly, this is the best implementation that we can do in the absence of any static information, and reduces to the standard applicative caching [11, 16].

4.2 Scheduling

The goal of scheduling is to find a timing function for each variable which maps each point in the variable's index space to a positive integer representing a relative execution time [9]. This mapping must respect the causality rule that states that if $A[p] \longrightarrow B[q]$, (variable A at index point p is dependent on variable B at index point q) then $t_A(p) > t_B(q)$, that is the computation of $A[p]$ follows the computation of $B[q]$. Multidimensional time can be used when affine schedules cannot be found in which case the causality rule states that $t_A(p) \succ t_B(q)$, where $\succ$ means "lexicographically greater" [10]. The scheduling problem in its full generality is undecidable [30]. Hence, we try to find timing functions within the restricted class of affine schedules, for which necessary and sufficient conditions exist [27, 29]. Extra flexibility is accorded by allowing different affine schedules for each variable [22].

For the forward substitution algorithm (Example 2, Figure 1), separate schedules for variables b and X are : $t_b(i,j) = 2j + 1$ and $t_x(i) = 2i$ for a minimum system latency of $2N-1, (0 \leq t \leq 2N-2)$. The inputs are all assumed to available at $t = 0$. Alternately, the schedules $t_b(i,j) = i + j$ and $t_x(i) = 2i$ also yield the same latency. Static analysis shows that the first schedule requires a one to many broadcast communication $X[i] \rightarrow b[i,j], i < j < N$, where as the second schedule allows $X[i]$ to be propagated to $b[i,j], i < j < N$ sequentially in a systolic array fashion. If the target architecture better supported row broadcasting than chained sequential communication, the first schedule would be chosen, otherwise, the second schedule would be chosen. Here, we choose the first. We introduce an intermediate local variable[3] x which is placed between the variable X and its computation. Then, the change of basis $(i \rightarrow 2i)$ is performed on variable x and $(i,j \rightarrow 2j+1, i)$ is performed on variable b to incorporate their respective schedules. The time index variable is renamed to t to remind us that it is now a temporal dimension. The resulting ALPHA code is[4] :

[3] Since X is an output, and we cannot do a change of basis on it without changing the system definition.

[4] The index Z is induced because the change of base is non–unimodular. It is a redundant temporal index $(= \lfloor \frac{t}{2} \rfloor)$, but necessary to keep the system affine.

```
system ForwardSubstitution ( parameter : { N | N>0 };
                             A : { i,j | 0< i<N; 0<=j<i } of integer;
                             B : { i   | 0<=i<N } of integer )
                  returns ( X : { i   | 0<=i<N } of integer );
  var
    x : {Z,t   | 2Z=t;   0<=t<=2N-2         } of integer;
    b : {Z,t,i | 2Z=t-1; 1<=t<=2i-1; i<=N-1 } of integer;
  let
    X[i] = x[i,2i];
    b[Z,t,i] = case
                 {Z,t,i|Z=0} : A[i,Z] * x[Z,t-1];
                 {Z,t,i|Z>0} : b[Z-1,t-2,i] + A[i,Z] * x[Z,t-1]
               esac;
    x[Z,i] =   case
                 {Z,t,N|Z=0} : B[Z];
                 {Z,t,N|Z>0} : B[Z] - b[Z-1,t-1,Z];
               esac;
  tel;
```

Causality can be quickly hand checked by observing that dependencies are now of the form `(t, ··· -> t` $-\alpha$`, ···)`.

4.3 Alignment and Virtual Allocation

The alignment problem is to find a mapping from the domain of each variable and computation to a common virtual domain for the entire system [7]. Thus the variables are aligned with respect to each other and are placed on a common grid. Alignment also affects communication and thus, can be done with a goal of reducing communication. When a computation is mapped to the same point as the variable on the left–hand side of the equation, it is called using the *owner computes rule* for aligning computation. Variables and computation can also be separately mapped to the common grid by using different mapping functions for computation and for variables.

For the forward substitution algorithm (Example 2, Figure 1), the variable x is a one–dimensional variable and b is a two–dimensional variable. During alignment, these two variables are placed on a common two dimensional grid. This is illustrated in figure 3. We map b to the virtual grid using the identity function $(i, j \rightarrow i, j)$. We place x on the virtual grid with b. The dependence $x[Z, t] \rightarrow b[Z - 1, t - 1, Z]$ suggests that $(Z, t \rightarrow Z, t, Z)$ mapping x to a diagonal would be a good change of basis, since it aligns x with the points on the virtual grid with which it communicates (see figure 3). After alignment, the resulting ALPHA program is:

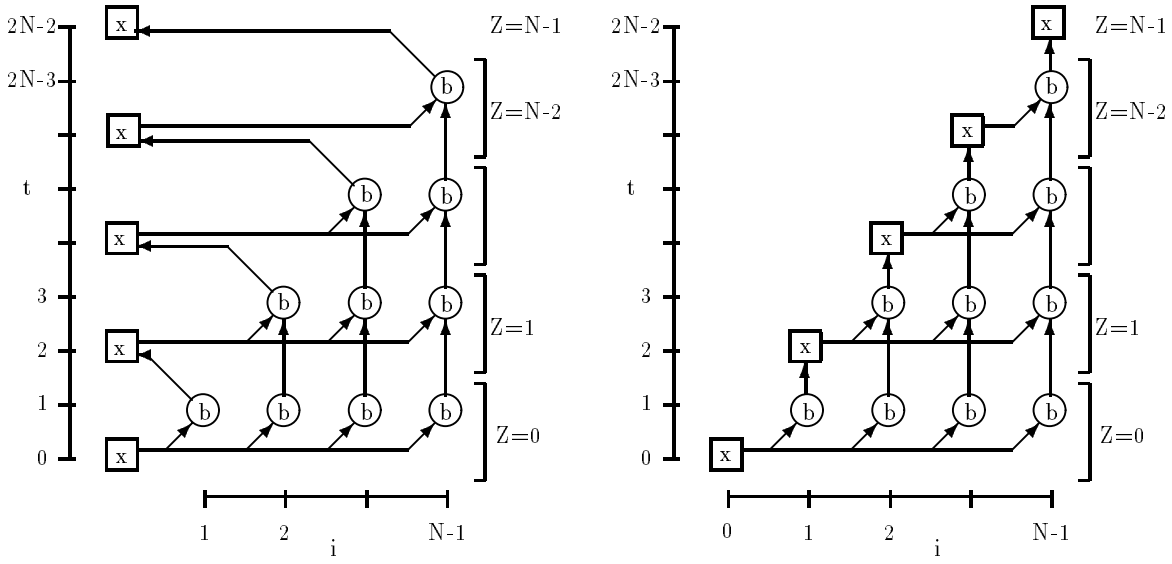


Figure 3: Alignment of variables in Forward Substitution

```
system ForwardSubstitution ( parameter : { N | N>0 };
                             A : { i,j | 0< i<N; 0<=j<i } of integer;
                             B : { i   | 0<=i<N } of integer )
                   returns ( X : { i   | 0<=i<N } of integer );
  var
    x : { Z,t,i|2Z=t;   t = 2i;     0<=i<=N-1 } of integer;
    b : { Z,t,i|2Z=t-1; 1<=t<=2i-1; 0<=i<=N-1 } of integer;
  let
    X[i] = x[i,2i,i];
    b[Z,t,i] = case
                 {Z,t,i|Z=0} : A[i,Z] * x[Z,t-1,Z];
                 {Z,t,i|Z>0} : b[Z-1,t-2,i] + A[i,Z] * x[Z,t-1,Z];
               esac;
    x[Z,t,i] = case
                 {Z,t,i|Z=0} : B[Z];
                 {Z,t,i|Z>0} : B[Z] - b[Z-1,t-1,i];
               esac;
  tel;
```

4.4 Partitioning and Physical Allocation

Partitioning is the problem of mapping variables and computations to physical processors [5]. The virtual grid is partitioned or tiled into groups which are to be mapped to physical processors. This partitioning has a profound effect on communication. It is therefore done so as to minimize communication while maximizing parallelism.

For the forward substitution algorithm (Example 2, Figure 1), the following new dependence table is generated:

```
{i    |0<=i<=N-1}          : X[i]     -> x[i,2i,i]       (1)
{Z,t,i|Z=0;   1<=i<=N-1}   : b[Z,t,i] -> A[i,Z]          (2)
{Z,t,i|Z=0; 1<=i<=N-1}     : b[Z,t,i] -> x[Z,t-1,Z]      (3)
{Z,t,i|1<=Z<=i-1; i<=N-1} : b[Z,t,i] -> b[Z-1,t-2,i]    (4)
{Z,t,i|1<=Z<=i-1; i<=N-1} : b[Z,t,i] -> A[i,Z]          (5)
{Z,t,i|1<=Z<=i-1; i<=N-1} : b[Z,t,i] -> x[Z,t-1,Z]      (6)
{Z,t,i|Z=0}                : x[Z,t,i] -> B[Z]           (7)
{Z,t,i|1<=Z<=N-1}          : x[Z,t,i] -> B[Z]           (8)
{Z,t,i|1<=Z<=N-1}          : x[Z,t,i] -> b[Z-1,t-1,i]   (9)
```

Each dependence represents a potential communication. With static analysis, the amount and cost of the communication can be estimated [28]. This information can be used to choose an allocation function for the virtual array. Dependency 1 represents the writing of the result X. It would represent a communication with the host, or I/O system. Likewise, dependencies 2,5,7, and 8 represent the inputs to the system which are also communications with the host, or I/O system. That leaves dependencies 3,4,6, and 9 as possible processor to processor communications. If the processor allocation function is chosen to be $p(Z,t,i) = i$ then dependencies 4 and 9 are between processor i and itself, and hence have no cost. That leaves communications 3 and 6 which are a set of N one to many broadcasts: $0 <= Z < N$: (from $x[Z, t-1, Z]$ to $\{Z < i <= N-1 : b[Z,t,i]\}$). Thus the projection $p(Z,t,i) = i$ is chosen. The transformed index is renamed p to represent the physical processor dimension. The resulting ALPHA code is similar to the last result, except the i-index is replaced with p in the equations for x and b.

4.5 Loop Nest Synthesis

This is the problem of converting recurrence equations which have been mapped to the same processor to loop nests, or sequences of loop nests. The result is code which resembles sequential imperative code encoded in ALPHA. This step is lumped in with code generation much of the time. However, we separate the steps of loop placement, computation of loop bounds, and the ordering of code from code generation since these steps can be done as transformations of an ALPHA program.

For the forward substitution algorithm (Example 2, Figure 1), the loop over processor space is factored out to be the outermost loop and the inner temporal loops are separated and ordered. The resulting ALPHA program is:

```
system ForwardSubstitution ( parameter : { N | N>0 };
                             A : { i,j | 0< i<N; 0<=j<i } of integer;
                             B : { i   | 0<=i<N } of integer )
                  returns ( X : { i   | 0<=i<N } of integer );
  var
    x : { Z,t,p|2Z=t;   t = 2p;     0<=p<=N-1 } of integer;
    b : { Z,t,p|2Z=t-1; 1<=t<=2p-1; 0<=p<=N-1 } of integer;
  let
    {Z,t,p | 0<=p<=N-1} :
      let
        {Z,t,p | t=0; Z=0; p=0}: x[Z,t,p] = B[0];
        {Z,t,p | t=1; Z=0; p>0}: b[Z,t,p] = A[p,0] * x[Z,0,0];
        {Z,t,p | 2<=t<2N-1; 1<=Z<N }:
          let
            {Z,t,p | 2Z=t; p=Z }:
                x[Z,t,p] = {Z,t,p|Z>0} : B[Z] - b[Z-1,t-1,p];
            {Z,t,p | 2Z=t-1; p>Z }:
                b[Z,t,p] = {Z,t,p|Z>0} : b[Z-1,t-2,p] + A[p,Z] * x[Z,t-1,Z];
          tel;
      tel;
    {p | 0<=p<N } :
        X[p] = x[p,2p,p];
  tel;
```

4.6 Code Generation

Code generation, as presented here, is translating code structures in an ALPHA program with equivalent code structures in the target language in order to create a valid target language program.

An important problem in doing code generation is the generation of variable declarations in the target language to replace the polyhedron based variable declarations in the ALPHA language. Certain optimizations may be done on an ALPHA program to reduce the amount of memory required to represent a variable. For instance, if it can be shown that for a certain dimension of the variable that the variable element lifetimes are disjoint, then that dimension can be projected out when allocating memory. For the remaining dimensions, a change of basis may be found which reduces the rectilinear embedding of the variable [39].

For the forward substitution algorithm (Example 2, Figure 1), rectilinear bounding boxes where found for each of the domains. By using lifetime analysis, local variable `b` was reduced from a vector to a scalar in each processor. The final resulting imperative data parallel code is:

```
ForwardSubstitution

declarations
  parameter N | N>0;
  processors proc[0..N-1];
  global
    int A[1..N-1][0..N-2], B[0..N-1], X[0..N-1], p;
  local
    int Z,x,b;
begin
  for {p | 0<=p<N }
    in proc[p]
      begin parallel
        if (p==0) x = B[0];
        if (p>0)  b = A[p,0] * proc[0].x;
        for {Z | 1<=Z<N}:
          begin
             if (p==Z) x = B[Z] - b;
             if (p>Z)  b = b + A[p,Z] * proc[Z].x;
          end;
        X[p] = x;
      end parallel;
end;
```

5 Analytical Tools Used

The ALPHA environment is a toolchest from which different transformations can be used to transform a program from its current state toward some target state. If a transformation is needed which does not exist, it can be written and easily integrated into the system. The ALPHA environment is build on top of a symbolic algebra system where the abstract syntax tree of an ALPHA program can be symbolically manipulated. For this, we employ MATHEMATICA [41] which supports imperative, functional, and rule-based programming paradigms built on top of a symbolic calculator. Transformations also rely heavily on the polyhedral library [38] which provides the capability for doing fundamental operations on polyhedra. This is a library of functions such as Image, Intersection, Difference and Union which operate on unions of convex polyhedra.

6 Discussion and Conclusions

In this paper, we have illustrated our approach for the compilation of ALPHA, a functional, data-parallel language which was originally developed for systolic array synthesis. We have built on the ideas of applicative caching and monolithic arrays, and showed how static analysis can be used to improve the efficiency of applicative caching by orders of magnitude. This analysis is directly extensible to parallelization for shared memory and distributed memory computers. We anticipate that it can enable the compiler to determine the grain size, and make decisions such as the throttling of parallelism when it does not

yield any performance gain. In effect, we aim to be able to generate code that is as efficient as parallel imperative code.

Recent work on loop parallelization has focused on extracting the exact data flow information from loop programs [8, 23]. In this context, the program is first converted into a system of affine recurrence equations. This is used as the intermediate representation which the compiler analyses to produce final parallel code. Since we start with a functional program already, we are able to avoid this (often expensive) first step. Moreover, functional programs are much easier to write and prove correct, are close to mathematical specifications and are easy to understand. A subtle point to note is that the scheduling problem is undecidable in general [30], and hence the naive demand driven code has to be a fall back position for our compiler. On the other hand, for a system of AREs obtained by analyzing a loop program it is guaranteed that the original lexicographic order is a valid (completely sequential) schedule.

Because of the elegance of functional languages, one would expect them to be naturally suited for numeric and scientific computing. One of the reasons that this has not come about, notwithstanding efforts such as SISAL and others [24, 25], seems to be because of efficiency, particularly in dealing with arrays. Typical compiler optimizations [2, 15, 26], such as common subexpression elimination depend on pattern matching and do not extend to interprocedural analysis. Since function application is *the* fundamental operation in program evaluation, this means that a lot of redundant computation cannot be avoided. Functional languages treat arrays either as incremental or as monolithic. The latter view is well suited to ALPHA and is also consistent with data parallel programming [12]. In spite of the expressive power of monolithic arrays, the problem of efficiency remains. The work presented here shows how functional programming can tap into the large body of work on imperative loop parallelization.

We feel that the research in three somewhat disparate areas—functional programming, loop parallelization and systolic array synthesis has shared many concerns, without building on each others' results. Functional languages have been clean but inefficient, imperative loop programs are dirty but efficient, and systolic arrays are overly specialized and often ignore the practical reality of partitioning, control generation, I/O, etc. We feel that there is much to be gained by cross fertilization of these issues across the different communities.

References

[1] C. Ancourt and F. Irigoin. Scanning polyhedra with DO loops. In *Third Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 39–50, ACM SIGPLAN, ACM Press, 1991.

[2] L. Augustsson. A compiler for lazy ML. In *ACM Symposium on Lisp and Functional Programming*, pages 218–227, ACM, Austin, TX, August 1984.

[3] J. Backus. Can programming be liberated from the von neumann style ? a functional style and its algrebra of programs. *Communications of the ACM*, 21(8):613–641, August 1978. 1978 Turing Award Lecture.

[4] R. S. Bird. Tabulation techniques for recursive programs. *Computing Surveys*, 12(4):403–419, Dec 1980.

[5] P. Boulet, A. Darte, T. Risset, and Y. Robert. *(Pen)–ultimate tiling ?* Technical Report 93-36, Ecole Normale Supérieure de Lyon, Nov 1993.

[6] M. Chen, Y. Choo, and J. Li. Crystal: theory and pragmatics of generating efficient parallel code. In B.K. Szymanski, editor, *Parallel Functional Languages and Compilers*, page Chapter 7, ACM Press, 1991.

[7] A. Darte and Y. Robert. *A Graph–Theoretic Approach to the Alignment Problem.* Technical Report 93-20, Ecole Normale Supérieure de Lyon, Jul 1993.

[8] P. Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–53, Feb 1991.

[9] Paul Feautrier. Some efficient solutions to the affine scheduling problem, Part I, One-dimensional Time. *International Journal of Parallel Programming*, 21(5), 1992.

[10] Paul Feautrier. Some efficient solutions to the affine scheduling problem, Part II, Multidimensional Time. *International Journal of Parallel Programming*, 21(6), 1992.

[11] D. P. Friedman, D. S. Wise, and M. Wand. Recursive programming through table lookup. In *Symposium on Symbolic and Algebraic Computation*, pages 85–89, ACM, New York, NY, 1976.

[12] G. Hains and X. Mullin, editors. *ATABLE 92: Second International Workshop on Array Structure*, DIRO, Univ de Montreal, Montreal, 1992.

[13] Philip J. Hatcher and Michael J. Quinn. *Data–Parallel Programming on MIMD Computers.* The MIT Press, Cambridge, Mass., 1991.

[14] P. Hudak and A. Bloss. The aggregate update problem in functional programming systems. In *ACM Symposium on Principles of Programming Languages*, pages 300–314, ACM, New Orleans, LA, January 1985.

[15] T. Johnsson. Efficient compilation of lazy evaluation. In *ACM Conference on Compiler Construction*, pages 58–69, ACM, Montreal, Canada, June 1984.

[16] R. M. Keller and M. R. Sleep. Applicative caching. *ACM Transactions on Programming Languages and Systems*, 8(1):88–108, January 1986.

[17] H. Le Verge. Reduction operators in Alpha. In D. Etiemble and J-C. Syre, editors, *Parallel Algorithms and Architectures, Europe*, pages 397–411, Springer Verlag, Paris, June 1992. See also [20].

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

[19] H. Le Verge, V. Van Dongen, and D. Wilde. La synthèse de nids de boucles avec la bibliothèque polyédrique. *RenPar'6*, Jun 1994.

[20] Hervé Le Verge. *Un environnement de tranformations de programmes pour la synthèse d'architectures régulières*. PhD thesis, Université de Rennes 1, Rennes, France, Oct 1992.

[21] Christophe Mauras. *ALPHA: un langage équationnel pour la conception et la programmation d'architectures parallèles synchrones*. PhD thesis, L'Université de Rennes I, IRISA, Campus de Beaulieu, Rennes, France, December 1989.

[22] Christophe Mauras, Patrice Quinton, Sanjay V. Rajopadhye, and Yannick Saouter. Scheduling affine parameterized recurrences by means of variable dependent timing functions. In S. Y. Kung and E. Swartzlander, editors, *International Conference on Application Specific Array Processing*, pages 100–110, IEEE Computer Society, Princeton, New Jersey, Sept 1990.

[23] D. Maydan, S. P. Amarsinghe, and M. Lam. Array data flow analysis and its use in array privatization. In *Principles of Programming Languages*, pages 2–15, ACM, January 1993.

[24] J.R. McGraw, S.K. Skedzielewski, S. Allan, and D. Grit. SISAL—streams and iteration in a single–assignment language. *Language Reference Manual, Version 1.2*, Jan 1985.

[25] Michael O'Boyle. *Program and Data Transformations for Efficient Execution on Distributed Memory Architectures*. PhD thesis, University of Manchester, Jan 1992.

[26] Simon Peyton Jones. *The Implementation of Functional Programming Languages. PHI Series in Computer Science, (editor, Hoare, C. A. R.)*, Prentice Hall, 1987.

[27] Patrice Quinton and Vincent Van Dongen. The mapping of linear recurrence equations on regular arrays. *Journal of VLSI Signal Processing*, 1(2):95–113, 1989.

[28] S. V. Rajopadhye. Analysis of affine communication specifications. In *IEEE Symposium on Parallel and Distributed Processing*, IEEE, Dallas, Texas, December 1993.

[29] Sanjay V. Rajopadhye. *Synthesis, Optimization and Verification of Systolic Architectures*. PhD thesis, University of Utah, Salt Lake City, Utah 84112, December 1986.

[30] Y. Saouter and P. Quinton. Computability of recurrence equations. *Theoretical Computer Science*, 114, 1993. (to appear: preliminary version available as IRISA TR-1203, April, 1990).

[31] Y. Saouter and P. Quinton. *Computability of Recurrence Equations*. Technical Report Internal Publication 521, IRISA, Rennes, France, Apr 1990. Also appeared as INRIA report number 1203.

[32] A. V. S. Sastry, W. Clinger, and Z. Ariola. Order-of-evaluation analysis for destructive updates in strict functional languages with flat aggregates. In *Functional Programming and Computer Architecture*, Springer Verlag, LNCS, Copenhagen, Jun 1993.

[33] D. A. Schmidt. Detecting global variables in denotational specifications. *ACM Transactions on Programming Languages and Systems*, 7(2):299–310, April 1985.

[34] Wolfgang Schreiner. *Parallel Functional Programming: An Annotated Bibliography.* Technical Report pfpbib.dvi.Z, Research Institute for Symbolic Computation (RISC), Johannes Kepler University, A-4040 Linz, Austria, May 1993.

[35] D. A. Turner. Recursion equations as a programming language. In Darlinton, Henderson, and Turner, editors, *Functional Programming and its Applications: an Advanced Course, 1981*, pages 1–28, Cambridge University Press, 1982.

[36] P. Wadler. A new array operation. In J. H. Fasel and R. M. Keller, editors, *Graph Reduction: Proceedings of a Workshop*, pages 328–335, Springer Verlag, LNCS 279, Santa Fe, NM, September 1986.

[37] M. R. Werth and P. Feautrier. On parallel program generation for massively parallel architectures. In M. Durand and F. El Dabaghi, editors, *High Performance Computing II*, pages 115–126, Elsevier Science Publishers B.V. (North Holland), 1991.

[38] D. Wilde. *A Library for Doing Polyhedral Operations.* Master's thesis, Oregon State University, Corvallis, Oregon, Dec 1993.

[39] D. Wilde and S. Rajopadhye. *Allocating memory arrays for polyhedra.* Technical Report Internal Publication 749, IRISA, Rennes, France, Jul 1993.

[40] M.E. Wolf and M. Lam. Loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):452–471, Oct 1991.

[41] Stephen Wolfram. *Mathematica: A System for Doing Mathematics by Computer, Second Edition.* Addison–Wesley Publishing Company, Inc., 1991.

[42] Jui-Hsiang Allan Yang. *Transformational Parallel-Program Derivations.* PhD thesis, Yale University, May 1993.