DISSERTATION

BEYOND SHARED MEMORY LOOP PARALLELISM IN THE POLYHEDRAL MODEL

Submitted by

Tomofumi Yuki

Department of Computer Science

In partial fulfillment of the requirements

For the Degree of Doctor of Philosophy

Colorado State University

Fort Collins, Colorado

Spring 2013

Doctoral Committee:

Advisor: Sanjay Rajopadhye

Wim Böhm
Michelle Strout
Edwin Chong

ABSTRACT

BEYOND SHARED MEMORY LOOP PARALLELISM IN THE POLYHEDRAL MODEL

With the introduction of multi-core processors, motivated by power and energy concerns, parallel processing has become main-stream. Parallel programming is much more difficult due to its non-deterministic nature, and because of parallel programming bugs that arise from non-determinacy. One solution is automatic parallelization, where it is entirely up to the compiler to efficiently parallelize sequential programs. However, automatic parallelization is very difficult, and only a handful of successful techniques are available, even after decades of research.

Automatic parallelization for distributed memory architectures is even more problematic in that it requires explicit handling of data partitioning and communication. Since data must be partitioned among multiple nodes that do not share memory, the original memory allocations cannot be directly used. One of the main contributions of this dissertation is the development of techniques for distributed memory parallel code generation with parametric tiling.

Our approach builds on important contributions to the polyhedral model, a mathematical framework for reasoning about program transformations. We show that many affine control programs can be uniformized only with simple techniques. Being able to assume uniform dependences significantly simplifies distributed memory code generation, and also enables parametric tiling. Our approach implemented in the AlphaZ system, a system for prototyping analyses, transformations, and code generators in the polyhedral model. The key features of AlphaZ are memory re-allocation, and explicit representation of reductions.

We evaluate our approach on a collection of polyhedral kernels from the PolyBench suite, and show that our approach scales as well as PLuTo, a state-of-the-art shared memory automatic parallelizer using the polyhedral model.

Automatic parallelization is only one approach to dealing with the non-deterministic nature of parallel programming that leaves the difficulty entirely to the compiler. Another approach is to develop novel parallel programming languages. These languages, such as X10, aim to provide highly productive parallel programming environment by including parallelism into the language design. However, even in these languages, parallel bugs remain to be an important issue that hinders programmer productivity.

Another contribution of this dissertation is to extend the array dataflow analysis to handle a subset of X10 programs. We apply the result of dataflow analysis to statically guarantee determinism. Providing static guarantees can significantly increase programmer productivity by catching questionable implementations at compile-time, or even while programming.

ACKNOWLEDGEMENTS

The years I spent as a student of Dr. Sanjay Rajopadhye were very exciting to say the least. The interactions we had over coffee were enjoyable, and had led to many ideas. I appreciate his help in many dimensions not limited to research but in a much broader context.

I am blessed to have Drs. Wim Böhm and Michelle Strout on my committee since my Master's thesis for every single examination that I went through. For every one of them, they have given valuable feedback through questions and comments.

The class on non-linear optimization by Dr. Edwin Chong has helped significantly in boosting my mathematical maturity. I will continue to look for applications of the optimization techniques we learned.

I am very thankful to Dr. Steven Derrien for introducing us to many interesting new ideas and Model-Driven Engineering. Without Steven and Model-Driven Engineering, we would not have advanced at this rate. The on-going collaboration with Steven has been very fruitful, and we will continue to work together.

We thank Dr. Dave Wonnacott and his students for using our tools, and giving us feedback. Without a body of external users, the tools would have been much less mature.

Members of Mélange, HPCM, and CAIRN have helped me throughout the years by being available for discussion, providing feedbacks on talks, and having fun together.

I appreciate the National Science Foundation, and Colorado State University for supporting my assistantships during the course of my study.

I thank my parents for their continuous support over many years, and their open-mindedness. I now realize that their scientific background and its reflection in how I was raised is a large contributor to my accomplishment.

Lastly, I would like to thank Mrs. Heidi Juran, my third grade teacher at an elementary school in Golden, Colorado. The great experiences I had back then have had so much influence to my later decisions, eventually leading up to my graduate study.

TABLE OF CONTENTS

Chapter 1

Introduction

Parallel processing has been a topic of study in computer science for a number of decades. However, it is only in the past decade that coarse grained parallelism, such as loop level parallelism, become more than a small niche. Until the processor manufacturers encountered the "walls" of power and memory [18, 56, 127], the main source of increase in compute power came from increasing the processor frequency; through an easy ride on the Moore's Law. Now the trend has completely changed, and multi-core processors are the norm. Compute power of a single core has stayed flat, or even decreased for energy efficiency, in the past 10 years. Suddenly, parallel programming became a topic of high interest in order to provide continuous increase in performance for the mass.

However, parallel programming is difficult. The moment parallelism is added to the picture, programmers now must think about many additional things. For example, programmers must first reason if a particular parallelization is legal. Parallelism introduces non-determinism and parallel bugs that arise from non-determinacy. Even if the parallelism is legal, is it efficiently parallelized?

One ultimate solution to this problem is automatic parallelization. If programmers can keep writing sequential programs, which they are now used to, and leave parallelization as a job of the compiler, then the increased compute power in the form of parallelism can easily be exploited.

However, automatic parallelization is very difficult due to many reasons. Given a program, the compiler must be able to ensure that the parallelized program still outputs the same result. The analyses necessary to provide such a guarantee is difficult, and currently known techniques are either over-approximate, by a large degree, or are only applicable to a restricted class of programs. In addition, a sequential program is already an over-specification in many cases. For example, sum of N numbers can be performed in $O(logN)$ steps in parallel, but in a sequential program, it takes $O(N)$ steps. Unless the compiler can reason about algebraic properties, associativity and commutativity in this case, it cannot exploit this parallelism.

An alternative approach for efficient and *productive* parallel programming is to develop new programming languages designed for parallel programming [7, 17, 80, 105, 116, 130]. Currently well accepted methods of parallel programming, such as OpenMP or Message Passing Interface (MPI), are essentially extensions to existing languages, like C or Fortran. On one hand, this allows reuse of an existing code base, and the learning curve may potentially be low. On the other hand, it requires both the compiler and programmers to cope with languages that were not originally designed for parallelism.

1.1 Scope of the Dissertation

The focus of this dissertation is on polyhedral programs, a class of programs that can be reasoned by a mathematical framework called the polyhedral model. For this class of programs—called Affine Control Loops (ACLs) [90], or sometimes called Static Control Parts (SCoPs) [13, 30]—the polyhedral model enables precise dependence analysis; statement instance-wise and array element-wise.

In the last decade, the polyhedral model has been shown to be one of the few successful approaches to automatic parallelization. Although the applicable class of programs are restricted, fully automatic parallelization has been achieved for polyhedral programs. Approaches based on polyhedral analyses are now part of production compilers [39, 83, 98], and many research tools [16, 19, 46, 64, 51, 78, 86] that use the polyhedral model have been developed.

In this dissertation, we address the following problems related to parallel processing and to polyhedral compilation.

- Unexplored design space of polyhedral compilers. Current automatic parallelizers based on the polyhedral model rarely re-consider the memory allocation of the original program. Memory allocations can have significant impact on performance by restricting applicable transformations. For example, the amount of parallelism may be reduced when memory allocation is untouched. Moreover, none of the polyhedral compilers take advantage of algebraic properties, such as those found in reductions.

- Distributed memory parallelization. The polyhedral model has been successful in automatically parallelizing for multi-core architectures with shared memory. The obvious next step is to target distributed memory systems, where the communication is now part of the programs, as opposed to implicit communication through shared memory.

  Distributed memory parallelization exposes two new problems that were conveniently hidden by the shared memory: communication and data partitioning. The compiler must reason about (i) which processors need to communicate, (ii) what values need to be communicated, and (iii) how to distribute storage among multiple nodes.

- Determinacy guarantee of a subset of X10 programs. Emerging parallel languages have a common goal of providing a *productive* environment for parallel programming. One of the important barriers that hinder productivity is parallel bugs arising from non-deterministic behaviors.

  The parallel constructs in X10 is more expressive than a commonly used form of parallelism: *doall* loops. Thus, previously approaches for race detection of parallel loop programs are not directly applicable.

1.2 Contributions

The polyhedral model plays a central role in all of our contributions that address the problems describe in the above. We present the following contributions in this dissertation:

- The AlphaZ system, a system for exploring the rich space of transformations and program manipulations available in the polyhedral model.
- Automatic parallelization targeting distributed memory parallelism with support for parametric tiling.
- Extension of Array Dataflow Analysis [30] to a subset of programs in the explicitly parallel high productivity language X10 [105].

The polyhedral model is now part of many tools and compilers [16, 19, 39, 46, 64, 51, 78, 83, 86, 98]. However, the design space explored by these tools is still a small subset of the space that can be explored within the polyhedral model. The AlphaZ system, presented in Chapter 3, aims to enlarge this subset, and to serve as a tool for prototyping analyses, transformations, and code generators. The key unique features of AlphaZ are (i) memory allocation, and (ii) reduction. Existing tools do not support altering memory allocation, or representing/transforming reductions. In addition, AlphaZ utilizes a technique from software engineering, called Model-Driven Engineering [34, 35].

Using the AlphaZ system, we have developed a set of techniques for generating distributed memory programs from sequential loop programs. The polyhedral model has been successful in automatically parallelizing for multi-core architectures with shared memory. The obvious next step is to target distributed memory systems, where the communication is now part of the programs, as opposed to implicit communication through shared memory. Since affine dependences can introduce highly complex communication patterns, we choose *not* to handle arbitrarily affine dependences.

We first question how "affine" affine loop programs are and show that most affine dependences can be replaced by uniform dependences (in Chapter 5.) The "uniform-ness" of affine programs is utilized in subsequent chapters that describe distributed memory code generation. In Chapter 6 we present techniques for automatically finding memory allocations for parametrically tiled programs. Memory re-allocation is a crucial step in generating distributed memory parallel programs. Chapter 7 combines the preceding chapters and present a method for automatically generating distributed memory parallel programs. The generated code differs from previously proposed methods [8, 15, 21, 97] by allowing parametrized tiling. We show that our distributed memory parallelization scales as well as PLuTo [16], the state-of-the-art polyhedral tool for shared memory automatic parallelization.

We have also extended the polyhedral model to handle a subset of X10 programs. Parallelism in X10 is expressed as asynchronous activities, rather than parallel loops. The polyhedral model has been used

for *doall* type parallelism, and cannot handle X10 programs. We present an extension to Array Dataflow Analysis [30] to handle a subset of X10 programs in Chapter 8. We show that the result of dataflow analysis can be used to provide race-free guarantees. The ability to statically verify determinism of a program region can greatly improve programmer productivity.

Finally, we summarize and conclude the discussion in Chapter 9 and present future directions.

Chapter 2

Background and Related Work

In this chapter we provide the necessary background of the polyhedral model, and discuss related work of our contributions. Section 2.1 covers basic concepts, such as polyhedral domains and affine functions, as well as the representations of program transformations, such as schedules and memory allocation, in the polyhedral model. In addition, Section 2.1.12 describes an equational view of polyhedral representations, specific to `AlphaZ` (and `MMAlpha` [64].) Another important background, tiling, its parameterization, and legality, is presented in Section 2.3.

We contrast `AlphaZ` with other tools and compilers that use the polyhedral model in Section 2.2. The related work on distributed memory code generation is discussed in Section 2.4.

2.1 The Polyhedral Model

The strength of the polyhedral model as a framework for program analysis and transformation are its mathematical foundations for two aspects that should be (but are often not) viewed separately: program *representation/transformation* and *analysis*. Feautrier [30] showed that a class of loop nests called Affine Control Loops (or Static Control Parts) can be represented in the polyhedral model. This allows compilers to extract regions of the program that are amenable to analyses and transformations in the polyhedral model, and to optimize these regions. Such code sections are often found in kernels of scientific programs, such as dense linear algebra, stencil computations, or dynamic programming. These computations are used in wide range of applications; climate modeling, engineering, signal/image processing, bio-informatics, and so on.

In the model, each instance of each statement in a program is represented as an *iteration point*, in a space called *iteration domain* of the statement. The iteration domain is described by a set of linear inequalities forming a convex polyhedron denoted as $\{z|\langle\text{constraints on } z\rangle\}$.

Dependences are modeled as pairs of affine function and domains, where the function represents the dependence between two iteration points, and the domain represents the set of points where the dependence exists. Affine functions are expressed as $(z \rightarrow z')$, where z' consists of affine expressions of z. Alternatively, dependences may be expressed as relations, sometimes called the dependence polyhedra, where functions and domains are merged into a single object. As a shorthand to write statement S depends on statement T, we also write $S[z] \rightarrow T[z']$.

2.1.1 Matrix Representation

Iteration domains and affine dependences may sometimes be expressed as matrices. A polyhedron is expressed as $Az+b \geq 0$ where A is a matrix, b is a constant vector, and z is a symbolic vector constrained by A and b. Similarly, an affine function is expressed as $f(z) = Az + b$ where A is a matrix, and b is a constant vector. Relations take the same form as domains in its matrix form.

2.1.2 Program Parameters

In the polyhedral model, the program parameters (e.g., size of inputs/outputs) are often kept symbolic. These symbolic values may also have a domain, and can also be used as part of the affine expressions. We follow the convention that capitalized index names denote implicit parameters and are not listed as part of z in both domains and functions. For example, when we write $\{i|0 \leq i \leq N\}$, N is some size parameter. Similarly $(i, j \rightarrow i, j + N)$ use an implicit parameter N.

2.1.3 Properties of Polyhedral Objects

One of the advantages of modeling the program using polyhedral objects is the rich set of closure properties that polyhedra and affine functions enjoy as mathematical objects. Preimage by function f, or image by its relational inverse f^{-1}, of a domain $\mathcal{D}$ is the set of points x such that $f(x) \in \mathcal{D}$. Polyhedral domains (unions of polyhedra) are closed under set operations. They are also closed under image by the relational inverse of affine functions, also called preimage. Because of this closure property, affine transformations are guaranteed to produce another polyhedra after its application.

In addition, a number of properties from linear algebra can be used to reason about the program. For some of the analyses in this paper, we use one class of such properties, namely the kernels of matrices, and by implication, of affine functions and domains. The kernel of matrix A, $\ker(A)$, is the set of vectors x such that $Ax = 0$. Note that if $\rho \in \ker(A)$ then $Az = A(z + \rho)$, so the space characterized by the kernel describes the set of vectors that do not affect the value of an affine function.

With an abuse of notation, we define the *kernel*s of domains and affine functions to be the respective kernels of the matrix that describes the linear part of the domain and affine functions. The kernel of domain $\mathcal{D}$ represented as $Ax + b \geq 0$ in matrix representation, is $\ker(A)$.

Another property used in the document is linearity spaces, of domains. The linearity space $\mathcal{H}_{\mathcal{D}}$ of domain $\mathcal{D}$ is the smallest affine subspace containing $\mathcal{D}$. The subspace is the entire space $\mathbb{Z}^N$, unless all points in $\mathcal{D}$ lie in a space of some lower dimension. In other words, if there are equalities in the domain $\mathcal{D}$, the equalities are what characterize the linearity space.

2.1.4 Uniform and Affine Dependences

A dependence is said to be uniform if the matrix A in the matrix representation is the identity matrix I. In other words, uniform functions are translations by some constant vector. Since the constant vectors are sufficient to characterize uniform dependences, they are referred as dependence vectors in this document.

2.1.5 Dependence vs Dataflow

In the literature of the polyhedral model, the word dependence is sometimes used to express flow of data, but in this dissertation, when we write and draw a dependence, the arrow is from consumer to producer. With an exception of dataflow vector, which is simply the negation of its corresponding dependence vector, we use dependences in this document.

2.1.6 Memory-Based Dependences

The results of array dataflow analysis are based on the values computed by instances of statements, and therefore do not need any notion of memory. As a consequence, program transformation using dataflow analysis results usually requires re-considering memory allocation of the original program. Most existing tools have made the decision to preserve the original memory allocation, and include memory-based dependences as additional dependences to be satisfied.

2.1.7 Lexicographical Order

Lexicographical ordering is used to describe the relation between two vectors. In this paper we use $\ll$ and $\gg$ to denote lexicographical ordering. Given two vectors z and z', $z \ll z'$ if

$$\exists k; \forall i < k, z_i = z'_i, z_k < z'_k$$

In words, z lexicographically precedes z' if some k-th element of z is less than z', and for all elements i that are before k, z_i and z'_i are equal.

Lexicographical ordering is the base notion of "time" in multi-dimensional polyhedra used in the polyhedral literature.

2.1.8 Polyhedral Reduced Dependence Graph

Polyhedral Reduced Dependence Graph (PRDG), sometimes called Generalized Dependence Graph, is a concise representation of dependences in a program. Each node of the PRDG represents a statement in the loop program, Nodes are connected with edges that represent dependences between statements. Nodes in PRDG have an attribute, its domain, which is the domain of the corresponding statement. Edges have two attributes, its domain and the dependence function; the pair of data that characterize a dependence. The

direction of the edge is the same as the dependence function, from the consumer to the producer. PRDG is a common abstraction of the dependences used in various analyses and transformations.

2.1.9 Scanning Polyhedra

After analyzing and transforming polyhedral representation of loop programs, an important step is to generate executable code in the end. The dominant form of such code generation is to produce loop nests that scan each point in the iteration domain once, and only once, in lexicographical order [13, 91]. The algorithm currently being used by most researchers was presented by Bastoul [13], which extends an earlier version by Quilleré [91], and implemented as the Chunky Loop Generator (CLooG).

2.1.10 Schedule

Schedules in the polyhedral model are specified as multi-dimensional affine functions. These functions map statement domains to another domain, where its lexicographic order denotes the order in which statement instances are executed [31, 32]. These affine schedules encompass wide range of loop transformations, such as loop permutation, fusion, fission, skewing, tiling, and so on. Not only that they represent transformations given above, compositions of loop transformations are also handled as compositions of affine functions.

2.1.11 Memory Allocation

There are a number of techniques for memory allocation in the polyhedral model [25, 69, 90, 112, 115]. Existing techniques all compute memory allocation of a statement; a node in the PRDG. Most techniques require schedules to be given before computing memory allocations.

Allocations are expressed as pseudo-projections, a combination of affine functions and modulo factors. The affine function, usually many-to-one, represents a projection that maps iteration points to (virtual) array elements. All points in the kernel of the projection are mapped to the same element, and hence share the same memory location. Modulo factors are specified for each dimension of the mapping, and when specified, memory locations are reused periodically using modulo operations.

For example, $(i, j \rightarrow i, j) \bmod [2, -]$ is a memory allocation where two points $[i, j]$ and $[i', j']$ share the same location in memory if $i \bmod 2 = i' \bmod 2 \wedge j = j'$.

2.1.12 Polyhedral Equational Model

The `AlphaZ` system adopts an *equational* view, where programs are described as mathematical equations using the `Alpha` language [76]. After array dataflow analysis of an imperative program, the polyhedral representation of the flow dependences can be directly translated to an `Alpha` program. In addition, `Alpha` has reductions as first-class expressions [62] providing a richer representation.

We believe that application programmers (i.e., non computer scientists), can benefit from being able to program with equations, where performance considerations like schedule or memory remain unspecified. This enables a separation of what is to be computed, from the mechanical, implementation details of *how* (i.e., in which order, by which processor, thread and/or vector unit, and where the result is to be stored.)

To illustrate this, consider a Jacobi-style stencil computation, that iteratively updates a 1-D data grid over time, using values from the previous time step. A typical `C` implementation would use two arrays to store the data grid, and update them alternately at each time step. This can be implemented using modulo operations, pointer swaps, or by explicitly copying values. Since the former two are difficult to describe as affine control loops, the Jacobi kernel in `PolyBench/C 3.2` [84] uses the latter method, and the code (`jacobi_1d_imper`) looks as follows:

```
for (t = 0; t < T; t++)
   for (i = 1; i < N-1; i++)
      A[i] = foo(B[i-1] + B[i] + B[i+1]);
   for (i = 1; i < N-1; i++)
      B[i] = A[i];
```

When written equationally, the same *computation* would be specified as:

$$A(t,i) = \begin{cases} t = 0: & B_{\texttt{init}}(i); \\ t > 0 \leq i < N-1: & \texttt{foo}(A(t-1,i-1), A(t-1,i), A(t-1,i+1)); \\ t > 0 = i: & A(t-1,i); \\ t > 0 \wedge i = N-1: & A(t-1,i); \end{cases}$$

where A is defined over $\{t,i|0 \leq t < T \wedge 0 \leq i < N\}$, and B_{init} provides the initial values of the data grid. Note how the loop program is already influenced by the decision to use two arrays, an implementation decision, not germane to the computation.

2.1.12.1 System of Recurrence Equations

The polyhedral model has its origin in analyses of System of Recurrence Equations (SREs), where a program is described as a system of equations, with no notion of schedule or memory [50]. A SRE is called System of Uniform Recurrence Equations (SURE), if the dependences consists only of uniform dependences. Similarly, if a system equations consists of affine dependences, it is called System of Affine Recurrence Equations (SARE).

The polyhedral representation of, and hence the affine control loops themselves, can be viewed as SREs using results of array dataflow analysis. The `Alpha` language is a superset of SAREs that in addition to an SRE, can represent reductions as first class objects.

2.1.12.2 Change of Basis

Change of Basis (CoB) is a transformation used mostly in the equational side of the polyhedral mode. CoB is a semantic preserving transformation used for multiple purposes. The transformation takes an affine

function T that admits a left inverse for all points in the domain to be transformed, T^{-1}, and a target statement/equation S, and transforms its domain by taking its image by T. Then, to preserve the original semantics, dependences in the program are updated with the following rules:

- All dependences f to S are replaced by $T \circ f$. Since S is transformed by T, composition with T is necessary to reach the same point as the original program.
- All dependences f from S are replaced by $f \circ T^{-1}$. Since S is transformed by T, its inverse is first applied to get back to the original space, and then f is applied to reach the same points as the original program.

CoB is used to change the *view* of the program without changing what is computed. The view may be its dependence patterns or shape of the domain and so on.

2.2 Polyhedral Compilation Tools

The polyhedral model has a long history, and there are many existing tools that utilize its power. Moreover, it is now used internally in the IBM XL compiler family [98]. We now contrast `AlphaZ` with such tools. The focus of our framework is to provide an environment to try many different ways of transforming a program. Since many automatic parallelizers are far from perfect, manual control of transformations can sometimes guide automatic parallelizers as we show later.

PLuTo

PLuTo is a fully automatic polyhedral source-to-source program optimizer tool that takes C loop nests and generates tiled and parallelized code [16]. It uses the polyhedral model to explicitly model tiling and to extract coarse grained parallelism and locality. Since it is automatic, it follows a specific strategy in choosing transformations.

Graphite

Graphite is an optimization framework for high-level optimizations that are being developed as part of `GCC` now integrated to its trunk [83]. Its emphasis is to extract polyhedral regions from programs that `GCC` encounters, a significantly more complex task than what research tools address, and to perform loop optimizations that are known to be beneficial.

`AlphaZ` is not intend to be full fledged compiler. Instead, we focus on intermediate representations that production compilers may eventually be able to extract. Although codes produced from our system can be integrated into a larger application, we do not insist that the process has to be fully automatic, thus expanding the scope of transformations.

PIPS

PIPS is a framework for source-to-source polyhedral optimization using interprocedural analysis [46]. Its modular design supports prototyping of new ideas by *developers*. However, the end-goal is an automatic parallelizer, and little control over choices of transformations are exposed to the user.

Polyhedral Compiler Collections

Polyhedral Compiler Collections (PoCC) is another framework for source-to-source program optimizations, designed to combine multiple tools that utilize the polyhedral model [86]. Like `AlphaZ`, POCC also seeks to provide a framework for developing tools like Pluto, and other automatic parallelizers. However, their focus is oriented towards automatic optimization of C codes, and they do not explore memory (re)-allocation.

MMAlpha

`MMAlpha` is another early system with similar goals to `AlphaZ` [64]. It is also based on the `Alpha` language. The significant differences between the two are that MMAlpha emphasizes hardware synthesis (therefore, considers only 1-D schedules, nearest-neighbor communication, etc.) It does not treat reductions as first class (the first thing an MMAlpha user does is to "serialize" reductions), and does no tiling. Moreover, it is based on Mathematica, and this limits its potential users by its learning curve and licensing cost. MMAlpha does provide memory reuse in principle, but in its context, simple projections that directly follow processor allocations are all that it needs to explore.

RStream

`Rstream` from Reservoir Labs performs automatic optimization of `C` programs [78]. It uses the polyhedral model to translate `C` programs into efficient code targeting multi-cores and accelerators. Vasillache et al. [119] recently gave an algorithm to perform a limited form of memory (re)-allocation (the new mapping must *extend* the one in the original program). In addition, RStream is also fully automatic, while our focus is on being able to express and explore different optimization strategies. Moreover, their tool is a commercial, closed-source system (although they do mention source-level collaborations are possible.)

Omega

The collection of tools developed as part of the Omega project [52, 51, 87, 110] together cover a larger subset of the design space than most other tools. The Omega calculator partially handles *uninterpreted function symbols*, which no other tools support. Their code generator can also re-allocate memory [110]. However, reductions are not handled by Omega tools.

CHiLL

`CHiLL` is a high-level transformation and parallelization framework using the polyhedral model [19]. `CHiLL` uses tools from the Omega project as its basis. It also allows users to specify transformation sequences through scripts. However, it does not expose memory allocation.

POET

`POET` is a script-driven transformation engine for source-to-source transformations [131]. One of its goals is to expose parameterized transformations via scripts. Although this is similar to `AlphaZ`, POET does not check validity of the transformations, and relies on external analysis to verify the transformations in advance.

2.3 Tiling

Tiling is a well known loop transformation that was originally proposed as a locality optimization [47, 96, 109, 124]. It can also be used to extract coarser grained parallelism, by partitioning the iteration space to tiles (blocks) of computation, some of which may run in parallel [47, 96].

2.3.1 Overview of Tiling

Tiling a d-dimensional loop nest usually results in $2d$-dimensional tiled loop nest. In the resulting loop nest, outer d-dimensional loop nest first visits all the tiles, and then another (inner) d-dimensional loop nest visits all points in a tile. Thus, tiling changes the execution order of the program, which may lead to better locality. We refer to the outer d-dimensional loops as the *tile loops*, and the inner d-dimensional loops as the *point loops*. In addition, the points visited by the tile loops are called *tile origins*, which are the lexicographically minimal points of the tiles. An example with 2D loop nest is illustrated in Figure 2.1.

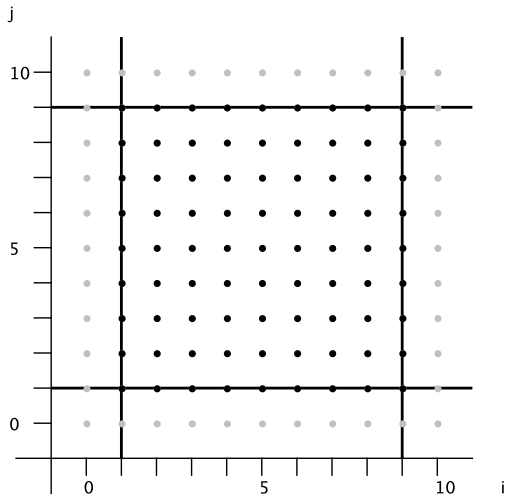
Another important notion related to tiling is the categorization of tiles into three types:

- Full Tile: All points in the tile are valid iterations.
- Partial Tile: Only a subset of the points in the tile are valid iterations.
- Empty Tile: No point in the tile is a valid iteration.

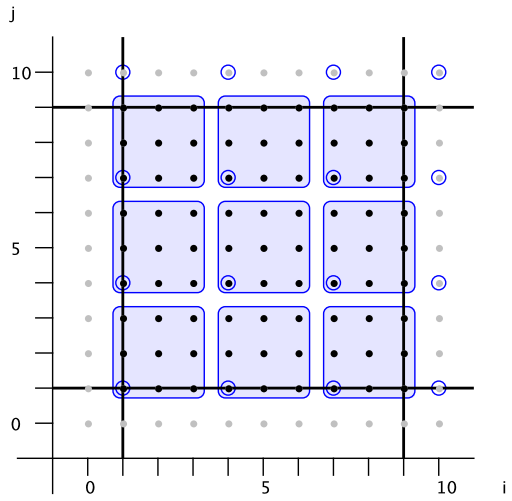
To reduce control overhead, one of the goals in tile loop generation is to avoid visiting empty tiles. Renganarayanan et al. [101] proposed what is called the *OutSet* that tightly over-approximates the set of non-empty tile origins, constructed as a syntactic manipulation of loop bounds. Similarly, they have presented *InSet* that exactly captures the set of full-tile origins.

```
for (i=1; i<10; i++)
    for (j=1; j<10; j++)
        ...
```

(a) Original loop nest



(b) Original iteration space



(c) Tiled iteration space

```
for (ti=1; ti<10; ti+=3)
    for (tj=1; tj<10; tj+=3)
        for (i=ti; i<min(ti+3,10); i++)
            for (j=tj; j<min(tj+3,10); j++)
                ...
```

(d) Tiled loop nest

Figure 2.1: Example of tiling with 2D loop nest. Tiling is a loop transformation that transform the original loop nest to tiled loop nest. Circled iteration point in each tile is the tile origin. The tile loop visits all points in a tile in lexicographic order before visiting points in the (lexicographically) next tile. Note that the sizes of the iteration space, as well as the tiles, are constants only for visualization purposes. Figures generated by Tiling Visualizer [103].

2.3.2 Non-Rectangular Tiling

The tiling used in the above is categorized as *rectangular tiling*, where the hyper-planes that define the tiles are along the canonic axes. In fact, most common tiling defined by hyper-planes that form hyper-parallelepipeds; n-dimensional generalization of parallelograms and parallelepipeds; can be tiled as rectangular tiles after skewing the iteration space [9].

Although tilings that cannot be expressed in this manner exits (e.g., [58]), rectangular tiling is currently the preferred method. This is mainly due to the lack (or quality) of code generation techniques for non-rectangular tiling. Rectangular tiling can be implemented efficiently as loop nests whereas other methods require more complicated control structure. In the rest of this dissertation, we refer to rectangular tiling when we mention tiling.

2.3.3 Parameterized Tiling

The choice of tile sizes significantly impacts performance. Numerous analytical models were developed for tile size selection (e.g., [22, 102, 99, 60]; see Renganarayana's doctoral dissertation [100] for a comprehensive study of analytical models.) However, analytical models are difficult to create, and can lose effectiveness due to various reasons, such as new architecture, new compilers, and so on. As an alternative method for predicting "good" tile sizes, more recent approaches employ some form of machine learning [93, 114, 132]. In these methods, machine learning is used to create models as platform evolves, and to avoid the need for creating analytical models to keep up with the evolution.

A complementary technique to the tile size selection problem is parameterization of tile sizes as run-time constants. If the tile sizes are run-time specified constants, instead of compile-time, code generation and compilation time can be avoided when exploring tile sizes. Tiling with fixed tile sizes; a parameter of the transformation that determines the size of tiles, can fit the polyhedral model. However, when tile sizes are parameterized, non-affine constraints are introduced and this falls out of the polyhedral formalism.

This led to the development of a series of techniques, beyond the polyhedral model, for parameterized tiled code generation [43, 44, 55, 54, 53, 101]. Initially, parameterized tiling was limited to perfectly nested loops and sequential execution of the tiles [55, 101]. These techniques were then extended to handle imperfectly nested loops [43, 54], and finally to parallel execution of the wave-front of tiles [44, 53].

DynTile [44] by Hartono et al., and D-Tiling [53] by Kim et al. are the current state-of-the-art of parameterized tiling for shared memory programs. These approaches both handle parameterized tiling of imperfectly nested loops, and its wave-front parallelization. Both of them manage the non-polyhedral nature of parameterized tiles by applying tiling as a syntactic manipulation of loops.

Our approach for distributed memory parallelization extends the ideas used in these approaches to handle parametric tiling with distributed memory.

2.3.4 Legality of Tiling

Legality of tiling is a well established concept defined over contiguous subsets of the schedule dimensions (in the RHS; scheduled space), also called *bands* [16]. These dimensions of the schedules are tilable, and are also known to be fully permutable.

The RHS of the schedules given to statements in a program all refers to the common schedule space, and have the same number of dimensions. Among these dimensions, a dimension is tilable if all dependences are not violated (i.e., the producer is not scheduled after the consumer, but possibly be scheduled to the same time stamp,) with a one-dimensional schedule using only the dimension in question. Then any contiguous subset of such dimensions forms a legal tilable band.

We call a subset of dimensions in an iteration space to be tilable, if the identity schedule is tilable for the corresponding subset.

2.4 Distributed Memory Parallelization

Although parallelism was not commonly exploited until the rise of multi-core architectures, it was used in the High Performance Computing field much before multi-cores. In HPC, more computational power is always demanded, and in many of the applications, such as simulating the climate, ample parallelism exists. As a natural consequence, distributed memory parallelization has been a topic of study for a number of decades.

When programming for distributed memory architectures, a number of problems that were not encountered in shared memory cases must be addressed. The two key issues that were not encountered in shared memory parallelization are *data partitioning* and *communication*.

Data Partitioning: When generating distributed memory programs starting from sequential programs, memory re-allocation must be re-considered. With shared memory, the same memory allocation was legal and efficient. However, with distributed memory, reusing the same memory allocation as the original program on all nodes multiplies the memory consumed by the number of nodes involved. Thus, it is necessary to re-allocate memory such that the total memory consumed is comparable to the original usage to provide scalable performance.

Communication: Since data are now local to each node, communication becomes necessary unless the computations are completely independent. In shared memory, communication is all taken care by the hardware or the run-time system. The only thing that is visible at the software are synchronization points, indicating points at which values written by a processor become available to others.

Note the above two problems, and also the partitioning of computation, which also arise in shared memory, are inter-related. The choice of data/computation partitioning can change what values are communicated and vice versa.

We distinguish our work from others in the following aspects:

- We support parametric tiling. None of the existing approaches handle parametric tiling for distributed memory parallelization.
- We explicitly manage re-allocation of memory. None of the existing polyhedral parallelizers for distributed memory even mention data partitioning. Instead, they use the same memory allocation as the original sequential program on all nodes.
- In contrast to those non-polyhedral approaches that handle data partitioning, we use the polyhedral machinery to:
  - apply loop transformations to expose coarse grained parallelism,
  - apply tiling, not performed by most approaches, and
  - in contrast to those that perform tiling, we handle imperfectly nested affine loops.
- We require at least one dimension to be uniform, or can be made uniform. This restriction
  - does not prevent us from handling most of `PolyBench` [84], and
  - simplifies communication and enables optimization of buffer usage, as well as overlap of communication with computation.

2.4.1 Polyhedral Approaches

Early ideas of distributed memory parallelization with polyhedral(-like) technique were presented by Amarasinghe [8]. Claßen and Griebl [21] later showed that, with polyhedral analysis, the set of values that needs to be communicated can be found. However, no implementation or experimental evaluation of their approach is available.

Bondhugula [15] has recently shown an approach that builds on previous ideas using the polyhedral formalism to compute values to be communicated. The proposed approach is more general than ours in that it handles arbitrarily affine dependences. However, the tile sizes must be compile-time constants. The author do not mention data partitioning, and it appears that the original memory allocation is used. In contrast, we handle parametric tile sizes, and we explicitly re-allocate memory to avoid excessive memory usage.

Kwon et al. [59] have presented an approach for translating OpenMP programs to MPI programs. They analyze shared data in the OpenMP program to compute the set of values that are used in a processor but

written in another. These values are communicated at the synchronization points in the original OpenMP parallelization as MPI calls. They handle a subset of affine loop nests where the set of values communicated do not change depending on the values of loop iterators surrounding the communication. Since parametrically tiled programs are not affine, they do not handle parametric tiling. The authors do not mention data partitioning other than input data, and their examples indicate that they do not touch the memory allocation.

2.4.2 Non-Polyhedral Approaches

The Paradigm compiler by Banerjee et al. [11] is a system for distributed memory parallelization. For regular programs, they apply static analysis to detect and parallelized independent loops, and then insert necessary communications.

Goumas et al. [37] proposed a system for generating distributed memory parallel code. Their approach is limited to perfectly nested loops with uniform dependences. They use non-rectangular, non-parameterized, tiling instead of skewing followed by rectangular tiling.

Li and Chen [71, 72] make a case that once computation and data partitioning is done, it is not difficult to insert communications that correctly parallelize the program in distributed memory. However, a naïve approach would result in point-to-point communication for each value used by other processors. They focus on finding reference patterns that can be implemented as aggregated communications.

As part of the `dHPF` compiler developed for High Performance Fortran [45], Mellor-Crummey et al. [79] use analysis on integer sets to optimize computation partitioning. In HPF, data partitioning is provided by the programmer, and it is the job of the compiler to find efficient parallelization based on the data partitioning. Although their approach is not completely automatic, they are capable of handling complex data and computation decompositions such as replicating computations.

Pandore [10, 36] is a compiler that take HPF(-like) programs as inputs, and produces distributed memory parallel code. Pandore uses a combination of static analysis and a run-time to efficiently manage pages of distributed arrays. Instead of finding out which values should be communicated as a block, the communication is always at the granularity of pages. Similarly, data partitioning is achieved by not allocating memory for pages not accessed by a node.

The main difference between these work and ours is the parallelization strategy. Most non-polyhedral approaches either find a parallelizable loop in the original program, or start from shared memory parallelizations with such information. Instead, we first tile the iteration space, and use specific properties from tiled iteration spaces in our distributed memory parallelization.

Chapter 3

The AlphaZ System

In this chapter, we present an open source polyhedral program transformation system, called `AlphaZ`, that provides a framework for prototyping analyses and transformations. `AlphaZ` is used for implementing the distributed code generator in Chapter 7. Memory re-allocation, and an extensible implementation of parameterized tiled code generator are critical elements of the distributed memory code generator, making `AlphaZ` an ideal system for its prototype implementation. Key features of `AlphaZ` are:

- Separation of implementation detail from the specification of computation. What needs to be computed is represented as Systems of Affine Recurrence Equations (SAREs), which takes the form of an equational language: `Alpha`.

  Execution strategies, such as schedules, memory allocations, and tiling, are specified orthogonally to the specification of computation itself.

- Explicit handling of reductions. Reductions; associative and commutative operator applied to a collection of values; are useful abstractions of the computation. In particular, `AlphaZ` implements a very powerful transformation that can reduce asymptotic complexity of programs using reductions [40].

Case studies to illustrate the potentials of memory re-allocation and reductions are presented in Chapter 4.

3.1 Motivations

The recent emergence of many-core architectures has given a fillip to automatic parallelization, especially through "auto-tuning" and iterative compilation, of compute- and data-intensive kernels. The *polyhedral model* is a formalism for automatic parallelization of an important class of programs. This class includes *affine control loops* which are the important target for aggressive program optimizations and transformations. Many optimizations, including loop fusion, fission, tiling, and skewing, can be expressed as transformation of polyhedral specifications. Vasillache et al. [85, 118] make a strong case that a polyhedral representation of programs is especially needed to avoid the blowup of the intermediate program representation (IR) when many transformations are repeatedly applied, as is becoming increasingly common in iterative complication and/or autotuning.

A number of polyhedral tools and components for generating efficient code are now available [16, 19, 44, 64, 51, 53, 78, 86]. Typically, they are source-to-source, and first extract a section of code amenable to polyhedral analysis, then perform a sequence of analyses and transformations, and finally generate output.

Many of these tools are designed to be fully automatic. Although this is a very powerful feature, and is the ultimate goal of the automatic parallelization community, it is still a long way away. Most existing tools give little control to the user, making it difficult to reflect application/domain specific knowledge and/or to keep up with the evolving architectures and optimization criteria. Some tools (e.g., CHiLL [19]) allow users to specify a set of transformations to apply, but the design space is not fully exposed.

In particular, few of these systems allow for explicit modification of the memory (data-structures) of the original program. Rather, most approaches assume that the allocation of values to memory is an inviolate constraint that parallelizers and program transformation systems must always respect. There is a body of work towards finding the "optimal" memory allocation [25, 69, 90, 112, 115]. However, there is no single notion of optimality, and existing approaches focus on finding memory allocation given a schedule or finding a memory allocation that is legal for a class of schedules. Therefore, it is critical to elevate data remapping to first-class status in compilation/transformation frameworks.

To motivate this, consider a widely accepted concept, *reordering*, namely changing the temporal order of computations. It may be achieved through tiling, skewing, fusion, or a plethora of traditional compiler transformations. It may be used for parallelism, granularity adaptation, or locality enhancement. Regardless of the manner and motivation, it is a fundamental tool in the arsenal of the compiler writer as well as the performance tuner.

An analogous concept is "*data remapping*," namely changing the memory locations where (intermediate as well as final) results of computations are stored. Cases where data remapping is beneficial have been noted, e.g., in array privatization [77] and the manipulation of buffers and "stride arrays" when sophisticated transformations like time-skewing and loop tiling are applied [125]. However, most systems implement it in an ad hoc manner, as isolated instances of transformations, with little effort to combine and unify this aspect of the compilation process into loop parallelization/transformation frameworks.

3.2 The AlphaZ System Overview

In this section we present an overview of the `AlphaZ` system,

`AlphaZ` is designed to manipulate `Alpha` equations, either written directly or extracted from affine control loops. It does this through a sequence of commands, written as a separate script. The program is manipulated through a sequence of transformations, as specified in the script. Typically, the final command in the script is a call to generate code (OpenMP parallel C, with support for parameterized tiling [44, 53]). The penultimate set of commands specify, to the code generator, the (i) schedule, (ii) memory allocation, and (iii) additional (i.e., tiling related) mapping specifications.

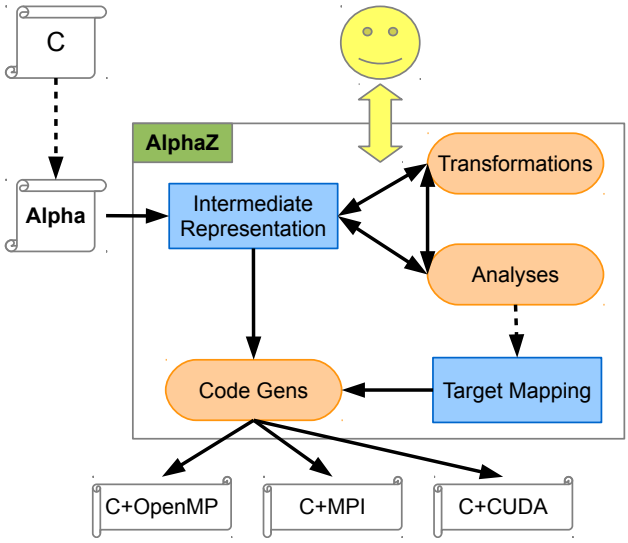


Figure 3.1: AlphaZ Architecture: The user writes an `Alpha` program (or extract it from `C`), and gives it to the system. The Intermediate Representation (IR) is analyzed and transformed with possible interactions with the user, and after high-level transformations, user specifies execution strategies called Target Mapping, some of which may also be found by the system through analyses. The specified Target Mapping and the IR is then passed to the code generator to produce executable code.

The key design difference from many existing tools is that `AlphaZ` gives the user full control of the transformations to apply. Our ultimate goal is to develop techniques for automatic parallelizers, and the system can be used as an engine to try new strategies. However, this has been the "ultimate goal" for many decades, well beyond the scope of a single doctoral dissertation. This allows for trying out new program optimizations that may not be performed by existing tools with high degree of automation. The key benefits for this are:

- Users can *systematically* apply sequences of transformations without re-writing the program by hand. The set of available transformations includes those involving memory re-mapping, and manipulating reductions.
- Compiler writers can *prototype* new transformations/code generators. New compiler optimizations may eventually be re-implemented for performance/robustness.

The input to the system is a language called `Alpha`, originally used in `MMAlpha`. As an alternative, we support automatic conversion of affine loop nests in `C` into `Alpha` programs. The PRDG extracted from loop programs using array dataflow analysis and the information about statement bodies in the original program is sufficient to construct the corresponding `Alpha` programs. The `Alpha` language may therefore be viewed as an Intermediate Representation (IR) of a compiler, with concrete syntax attached.

Figure 3.1 shows an overview of the system. We first describe the `Alpha` language in Section 3.3, and then present the Target Mapping in Section 3.4. Section 3.5 illustrates currently available code generators.

3.3 The Alpha Language

In this section we describe the Alpha language used in AlphaZ. The language we use is a slight, syntactic variant of the original Alpha [62]. In addition, an extension to the language to represent *while loops* and *indirect accesses*, called Alphabets has been proposed but is not fully implemented [94]. For the purposes of this paper, references to Alphabets should be considered the synonymous to Alpha.

3.3.1 Domains and Functions

Before introducing the language, let us first define notations for polyhedral objects, domains and functions. The textual representation of domains and functions resembles the mathematical notations with the following changes to use standard characters:

- `&&` denotes intersection and `||` denotes union
- $\rightarrow, \leq, \geq$ are written `->, <=, >=` respectively

We use the above textual representation when referring to a code fragment or when describing Alpha syntax.

When writing constraints for polyhedral domains, some short-hand notations are available. Constraints such as `a <= b` and `b <= c` can be merged as `a <= b <= c` if the constraints are "transitively aligned" ($<$ and $\leq$ or $>$ and $\geq$). If two indices share the same constraints, it can be expressed concisely by using a list of indices surrounded by parentheses (e.g., `a <= (b,c)` .)

3.3.1.1 Parameter Domains

Polyhedral objects may involve program parameters that represent problem size (e.g., size of matrices) as symbolic parameters. Except for where the parameters are defined, Alpha parser treats parameters as implicit indices. For example, a 1D domain of size N is expressed as {i|0<=i<N}, and not {N,i|0<=i<N}. Similarly, functions that involve parameters are expressed like (i->N-i), and not (N,i->N,N-i).

3.3.1.2 Index Names

Although textual representation of domains and functions use names to distinguish indices from each other, the system internally does not use the index names when performing polyhedral operations. The indices are distinguished from each other by *dimensions*. For example, domains:

- {i,j| 0<=i<N && 0<=j<M}
- {x,y| 0<=x<N && 0<=y<M}
- {j,i| 0<=j<N && 0<=i<M}
- {i,x| 0<=i<N && 0<=x<M}

are all equivalent, since constraints on the first dimension are always 0 to N, and constraints on the second dimension are always 0 to M. Similarly, the index names can be different for each polyhedron in a union of polyhedra. For example, `{i,j| 0<=i<N && 0<=j<M} || {x,y| 0<=x<P && 0<=y<Q}` is valid. The system does make an effort to preserve index names during transformations, but it cannot be preserved in general.

3.3.2 Affine Systems

An `Alpha` program consists of one or more affine systems. The high-level structure of an `Alpha` system is as follows:

```
affine <name> <parameter domain>
   input
      (<type> <name> <domain>;)*
   output
      (<type> <name> <domain>;)*
   local
      (<type> <name> <domain>;)*
   let
      (<name> = <expr>;)*
.
```

Each system corresponds to a System of Affine Recurrence Equations (SARE). The system consists of a name, a parameter domain, variable declarations, and equations that define values of the variables.

3.3.2.1 Parameter Domain

Parameter domain is a domain with indices and constraints that are true for all domains in the system. The indices in this domain are treated as program parameters mentioned above, and are implicitly added to all domains in the rest of the system.

3.3.2.2 Variable Declaration

Variable declarations specify the type and domain of each variable. We currently support the following types `int`, `long`, `float`, `double`, `char`, `bool`. The specified domain should have a distinct point for each value computed throughout the program, including intermediate results. It is important not to confuse domains of variables with memory, but rather as simply the set of points where the variable is defined. Some authors my find is useful to view this as single assignment memory allocation, where every memory location can only be written once.[1] Each output and local variable may correspond to a statement in a loop program, or an equation in an SRE. The body of the statement/equation is specified as `Alpha` expressions following the `let` keyword.

---

[1]We contend that so called, "single assignment" languages are actually zero-assignment languages. Functional language compilers almost always reuse storage, so nowhere does it make sense to use the term "single" assignment.

Table 3.1: Expressions in Alpha.

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

3.3.2.3 External Functions

External functions may additionally be declared in the beginning of an `Alpha` program. External function declarations take the form of C function prototypes/signatures with scalar inputs and outputs. Declared external functions can be used as point-wise operators, and are assumed to be side effect free.

3.3.3 `Alpha` Expressions

Table 3.1 summarizes expressions in `Alpha`. Expressions in `Alpha` also have an associated domain computed from the leaf (either constants or variables, where the domain is defined on its own) using domains of its children. These domains denote where the expression is defined and could be computed. Domain $\mathcal{D}_P$ in the table above, shown as the domain of constants and index expressions, is the parameter domain. These expressions can be evaluated for the full universe, and thus its expression domain is the intersection of universe with the parameter domain.

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 is either provided as input or given by an equation; in the latter case, it is the value, at z, of the expression on its RHS.
- an operator expression is the result of applying `op` on the values of its arguments at z. `op` is an arbitrary, strict point-wise, single valued functions. Also note that external functions are like user-defined operators.
- a case expression is the value at z of that 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.

- the dependence expression $f@E$ is the value of E at $f(z)$. The dependence expression in our variant of `Alpha` use function joins instead of compositions. For example, $f@g@E$ is the value of E at $g(f(z))$, whereas the original `Alpha` language defined by Mauras used $E.g.f$.

- the index expression `val`(f) is the value of f evaluated at point z.

- `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 $\oplus$.

It is important to note that the restrict expression only affects the domain, and not what is computed for a point. This expression is used in various ways to specify the range of values being computed for an equation. In addition, identity dependence is assumed for variable expressions without a surrounding dependence expression. Similarly, function to zero-dimensional space from the surrounding domain is assumed for constant expressions.

3.3.3.1 Reductions in Alpha

Reductions, associative and commutative operators applied to collections of values, are explicitly represented in the intermediate representation of AlphaZ. Reductions often occur in scientific computations, and have important performance implications. For example, efficient implementations of reductions are available in OpenMP or MPI. Moreover, reductions represent more precise information about the dependences, when compared to chains of dependences.

The reductions are expressed in the following form as `reduce`$(\oplus, f_p, \texttt{Expr})$, where *op* is the reduction operator, f_p is the projection function, and E is the expressions/values being reduced. The projection function f_p is an affine function that maps points in $\mathbb{Z}^n$ to $\mathbb{Z}^m$, where m is usually smaller than n (sof f is many-to-one mapping.) When multiple points in $\mathbb{Z}^n$ are mapped to a same point in $\mathbb{Z}^m$, the values of `Expr` at those points are combined using the reduction operator. For example, commonly used mathematical notations such as $X_i = \sum_{j=0}^{n} A_{i,j}$ is expressed as $X(i) =$ `reduce`$(+, (i, j \rightarrow i), A(i, j))$. This is more general than mathematical notations, allowing us to concisely specify reductions with non-canonic projections, such as $(i, j \rightarrow i + j)$.

3.3.3.2 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 [26]. The context domain of an expression E is computed from its domain and the context domain of its parent.

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

- $\mathcal{D}_{\mathtt{V}} \cap \mathcal{D}_{\mathtt{E}}$ if the parent is an equation for variable `V`.
- $f(\mathcal{X}_{E'})$ if E' is $E.f$.
- $f_p^{-1}(\mathcal{X}_{E'}) \cap \mathcal{D}_E$ if E' is reduce$(\oplus, f_p, E)$.
- $\mathcal{X}_{\mathtt{E}'} \cap \mathcal{D}_{\mathtt{E}'}$ if the parent $\mathtt{E}'$ 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.

3.3.4 Normalized `Alpha`

`Alpha` programs can become difficult to read, especially as program transformations are composed, and may have complicated expressions such as case or if expressions.

For example, consider the equation below (drawn from [62]).

```
U = case
      {i,j|j==0} : X;
      {i,j|j>=1} : (i,j->i+j)@(Y+Z)
                 * case
                     {i,j|i==0 && j>0} : W1;
                     {i,j|i>=1 && j>0} : W2;
                   esac;
       esac;
```

it would be much more readable if it were rewritten as:

```
U = case
      {i,j|j==0} : X;
      {i,j|i==0 && j>=1} : ((i,j->i+j)@Y + (i,j->i+j)@Z) * W1;
      {i,j|i>=1 && j>=1} : ((i,j->i+j)@Y + (i,j->i+j)@Z) * W2;
    esac;
```

Note how the case expressions are now "flattened." This flattening is the result of a transformation called *normalization*, as proposed originally by Mauras [76]. Normalized programs are usually easier to read since the branching of the cases are only at the top-level expression, and the reader does not have to think about restrict domains at multiple levels of case expressions. The important properties of normalized `Alpha` programs are:

- Case expressions are always the top-level expression of equations or reductions, and there is no nesting of case expressions.
- Restrictions, if any, are always just inside the case, and are also never nested. The expression `inside` a restriction has neither case nor restriction, but is a simple expression consisting of point-wise operators and dependence expressions.
- The child of dependence expressions are either a variable, a constant, or a reduce expression.

3.3.4.1 Normalization Rules

The following rules are used to normalize `Alpha` programs [76]. As a general intuition, restrict expressions are taken higher up in the AST, while dependence expressions are pushed down to the leaves.

1. $f@E \Rightarrow E$, if $f(z) = z$; Eliminating identity dependences.
2. $f@(E_1 \oplus E_2) \Rightarrow (f@E_1) \oplus (f@E_2)$; Distribution of dependence expressions.
3. $(\mathcal{D} : E_1) \oplus E_2 \Rightarrow \mathcal{D} : (E_1 \oplus E_2)$; Promotion of restrict expressions. Since $E_1 \oplus E_2$ is only defined for the set of points where both E_1 and E_2 are defined, restrict expressions can be applied to both.
4. $E_1 \oplus (\mathcal{D} : E_2) \Rightarrow \mathcal{D} : (E_1 \oplus E_2)$; Same as above.
5. $f_2@(f_1@E) \Rightarrow f@E$, where $f = f_1 \circ f_2$; Function composition.
6. $f_2@val(f_1) \Rightarrow val(f)$, where $f = f_1 \circ f_2$; Function composition involving index expressions.
7. $\mathcal{D}_1 : (\mathcal{D}_2 : E) \Rightarrow \mathcal{D} : E$, where $\mathcal{D} = \mathcal{D}_1 \cap \mathcal{D}_2$; Nested restrictions are equivalent to one restriction by the intersection of the two.
8. $case\, E_1^1; \ldots case\, E_1^2; \ldots E_n^2;\ esac \ldots E_m^1;\ esac \Rightarrow case\, E_1^1; \ldots E_1^2; \ldots E_n^2; \ldots E_m^1;\ esac$; Flattening of nested case expressions.
9. $E \oplus (case\, E_1; \ldots E_n;\ esac) \Rightarrow case\, (E \oplus E_1); \ldots (E \oplus E_n);\ esac$;
   Distribution of point-wise operations.
10. $(case\, E_1; \ldots E_n;\ esac) \oplus E \Rightarrow case\, (E_1 \oplus E); \ldots (E_n \oplus E);\ esac$; Same as above.
11. $f@(case\, E_1; \ldots E_n;\ esac) \Rightarrow case\, (f@E_1); \ldots (f@E_n);\ esac$;
    Distribution of dependence expressions.
12. $\mathcal{D} : (case\, E_1; \ldots E_n;\ esac) \Rightarrow case\, \mathcal{D} : E_1; \ldots \mathcal{D} : E_n;\ esac$;
    Distribution of restrict expressions.

13. $f@(if\ E_1\ then\ E_2\ else\ E_3) \Rightarrow if\ (f@E_1)\ then\ (f@E_2)\ else\ (f@aE_3)$;
    Distribution of dependence expressions.

14. $if\ (\mathcal{D} : E_1)\ then\ E_2\ else\ E_3 \Rightarrow \mathcal{D} : (if\ E_1\ then\ E_2\ else\ E_3)$; Promotion of restrict expressions. If-then-else expressions are only defined for the set of points where E_1, E_2, and E_3 are defined.

15. $if\ E_1\ then\ (\mathcal{D} : E_2)\ else\ E_3 \Rightarrow \mathcal{D} : (if\ E_1\ then\ E_2\ else\ E_3)$; Same as above.

16. $if\ E_1\ then\ E_2\ else\ (\mathcal{D} : E_3) \Rightarrow \mathcal{D} : (if\ E_1\ then\ E_2\ else\ E_3)$; Same as above.

17. $if\ (case\ E_1^1; \ldots\ E_n^1; esac)\ then\ E_2\ else\ E_3$
    $\Rightarrow case\ (if\ E_1^1\ then\ E_2\ else\ E_3); \ldots\ (if\ E_n^1\ then\ E_2\ else\ E_3);\ esac$;
    Distribution of if-then-else into case expressions.

18. $if\ E_1\ then\ (case\ E_1^2; \ldots\ E_n^2; esac)\ else\ E_3$
    $\Rightarrow case\ (if\ E_1\ then\ E_1^2\ else\ E_3); \ldots\ (if\ E_1\ then\ E_n^2\ else\ E_3);\ esac$; Same as above.

19. $if\ E_1\ then\ E_2\ else\ (case\ E_1^3; \ldots\ E_n^3; esac)$
    $\Rightarrow case\ (if\ E_1\ then\ E_2\ else\ E_1^3); \ldots\ (if\ E_1\ then\ E_2\ else\ E_n^3);\ esac$; Same as above.

3.3.5 Array Notation

For readability, an abbreviated notation is used for dependence expressions in parts of this dissertation. Since the examples we encounter are normalized, the parent of a variable expression is always a dependence node. For example, let A be a variable with one-dimensional domain, and it is used by another expression with 3D domain. Then the variable must be accessed as $A.f$, where f is an affine function from $\mathbb{Z}^3$ to $\mathbb{Z}^1$. For example, $A.(i, j, k \rightarrow k)$ is an access to a one-dimensional variable A from a 3D space by the dependence function $(i, j, k \rightarrow k)$.

However, when the index names are unambiguous from the context, we use the array notation and only write the RHS of the function. For the above example, the corresponding expression in array notation is $A[k]$ when it is clear from the "context" that the indices for 1st to 3rd dimensions are named i, j, k.

Array notation was created to allow dependence to variables resemble array accesses. In addition to dependences, there are other syntactic conveniences that rely on context information. These context-sensitive syntax are collectively called array notations. Other array notations in `Alpha` are:

- Index Expressions may use array notations similar to dependences. (e.g., `val`$(i, j, k \rightarrow i - j + 10)$ may be written as $[i - j + 10]$)
- Restrict Expressions may omit the index names in its restrict domain. Index names of restrict domains must either be fully given or fully omitted, and names from the context are used when omitted. (e.g., {`|0<=i<N`}).

- Reduce Expressions may specify their projection function using array notation. Array notation for projection functions is only applicable when the function is canonic. With canonic projections, index names for new dimensions may be specified with array notation. For example, `reduce(+, (i,j->i), ...)` may be written as `reduce(+, [j], ...)` .

Context of array notations are either defined by the LHS of the surrounding equation, or by a surrounding reduce expression. In the LHS of the equation, the variable name may be accompanied with a list of index names. When index names are given in the LHS, then those names are treated as the context in its RHS expressions. However, this context may be over written by a reduce expression. When a reduce expression is encountered, all its children will now be in a different context, defined by the LHS of the projection function. New index names will be simply appended if the projection function was specified with array notation.

For example, consider the following equation:

```
A[i,j] = X[i,j] + reduce(+, (x,y,z->x,z), Y[x-y+z]);
```

Access to `X` uses the context defined by the LHS of the equation, but access to `Y` uses the context defined by the reduce expression.

3.3.6 Example

We take a simple computation, matrix multiplication to illustrate basic syntax of the language. Matrix multiplication using reduction is written as follows in alphabets:

```
affine matrix_product {N|N>0}
  input
     double A,B {i,j|0<=(i,j)<N};
  output
     double C {i,j|0<=(i,j)<N};
  let
     C = reduce(+, (i,j,k->i,j), (i,j,k->i,k)@A * (i,j,k->k,j)@B);
.
```

Note that we only use one program parameter in the above example to avoid clutter, making it a square matrix multiplication.

Matrix multiplication can be written as follows without using reductions:

```
affine matrix_product {N|0<N}
   input
      double A,B {i,j|0<=(i,j)<N};
   output
      double C {i,j|0<=(i,j)<N};
   local
      double temp_C {i,j,k|0<=(i,j,k)<N};
   let
      C = (i,j,k->i,j,N-1)@temp_C;
      temp_C = case
```

```
            {i,j,k|k==0} : (i,j,k->i,k)@A * (i,j,k->k,j)@B;
            {i,j,k|k> 0} : (i,j,k->i,k)@A * (i,j,k->k,j)@B
                           + (i,j,k->i,j,k-1)@temp_C;
        esac;
.
```

Without reductions, the program must explicitly specify dependences for accumulation of the result matrix C. In the above program, accumulation is performed in a local variable `temp_C`.

The matrix multiplication example in array notation is written as follows:

```
affine matrix_product {N|0<N}
   input
      double A,B {i,j|0<=(i,j)<N};
   output
      double C {i,j|0<=(i,j)<N};
   let
      C = reduce(+, [k], A[i,k] * B[k,j]);
.

affine matrix_product {N|0<N}
   input
      double A,B {i,j|0<=(i,j)<N};
   output
      double C {i,j|0<=(i,j)<N};
   local
      double temp_C {i,j,k|0<=(i,j,k)<N};
   let
      C[i,j] = temp_C[i,j,N-1];
      temp_C[i,j] = case
             {|k==0} : A[i,k] * B[k,j];
             {|k> 0} : A[i,k] * B[k,j] + temp_C[i,j,k-1];
          esac;
.
```

3.4 Target Mapping: Specification of Execution Strategies

In this section we describe the Target Mapping (TMap) for specifying execution strategies. TMap consists of three main specifications, schedule to define when to compute, processor allocation for where to compute, and memory allocation for where to store the results. We will use a simple 3-point stencil computation with 1D data, shown in Figure 3.2, as an example in this section.

TMap is used for exploring transformations described as the combination of schedule, processor allocation, and memory allocations. In addition to these main axes, additional specifications such as tiling is also specified in TMap. The main specifications are specified per "variable" in alphabets programs, which corresponds to statements in ACLs.

```
affine jacobi1D {N,T|N>0 && T>0}
   input
      double Ain {i|0<=i<N}
   local
      double A {t,i|0<=i<N && 0<=t<=T}
   output
      double Aout {i|0<=i<N}
   let
      A[t,i] = case
                 {t==0} : Ain[i,j]
                 {t>0 && 1<=i<N-1} :
                   (A[t-1,i] + A[t-1,i-1] + A[t-1,i+1]) * 0.333333;
                 {t>0 && i==0} || {t>0 && i==N-1} : A[t-1,i];
               esac;
     Aout[i] = A[T,i];
.
```

Figure 3.2: `Alpha` specification of 3-point Jacobi stencil.

3.4.1 Space-Time Mapping

We use multi-dimensional affine functions to jointly represent schedules and processor allocation. We call this multi-dimensional mapping the space-time (ST) mapping. Space-time mapping maps domains of variables to others, where each dimension will eventually corresponds to a loop in the generated code.

The following restrictions apply to the given mapping:

- The ST maps for all variables must have the same number of destination dimensions. Since all statements must be placed relative to each other, all statements must be scheduled in a common space.
- The mapping must be bijective, so that the statements are executed for the appropriate point in its domain.

Space-time mapping can be seen as multi-dimensional schedule [32], and is sometimes referred to as a full-dimensional schedule. In this dissertation, the non-parallel dimensions of the space-time mapping are referred to as schedules.

3.4.1.1 Schedule

We first describe the case when the entire space-time mapping is the schedule—sequential programs. A large number of loop transformations can be expressed with multi-dimensional affine schedules, and they are used by most program transformation tools using the polyhedral model.

For example, the schedule `(i,j->j,i)` can be viewed as a loop permutation when compared to another schedule `(i,j->i,j)`. Similarly, the schedule `(t,i->t,t+i)` corresponds to loop skewing, skewing i loop by t.

How a user may specify the above schedule for loop skewing in our script is illustrated below:

```
prog = ReadAlphabets("jacobi1D.ab");
SetSTMap(prog, "A", "(t,i->t,i+t)");
```

`SetSTMap` command is used to specify space-time mapping for each variable.

3.4.1.2 Processor Allocation

In AlphaZ, processor allocation is part of the space-time mapping, and is distinguished from the time component of the ST mapping by annotations given to dimensions. After specifying the space-time mapping, certain dimensions may be flagged as parallel using `SetParallel` command.

For example, we may first use the SetSTMap command to specify lexicographic schedule, and then `SetParallel` command to flag the inner dimensions as parallel. Dimensions of ST mappings are specified as integer index starting from 0.

The script to specify ST mapping for executing the inner loop of Jacobi 1D stencil is the following:

```
prog = ReadAlphabets("jacobi1D.ab");
SetSTMap(prog, "A", "(t,i->t,i)");
SetSTMap(prog, "Aout", "(i->T-1,i)");
SetParallel(prog, 1);
```

3.4.1.3 Ordering Dimensions

In addition to sequential and parallel dimensions, another possible dimension type is called the ordering dimension. When generating imperfect loop nests from polyhedral representation, the common practice is to use additional dimensions, with constant values, to denote ordering of the loops. For example, given two statements `S1` and `S2` with the same domain $\{i, j | 0 \leq (i, j) < N\}$, generating code with schedules $\theta_{S1} = (i, j \rightarrow i, 0, j)$ and $\theta_{S2} = (i, j \rightarrow i, 1, j)$ will produce:

```
for (i=0; i < N; i++)
   for (j=0; j < N; j++)
      S1
   for (j=0; j < N; j++)
      S2
```

If, only the j loop surrounding $S1$ is to be executed in parallel, the user may specify the values of ordering dimensions that distinguish the target loop as an optional argument to the `SetParallel` command as follows:

```
SetSTMap(prog, "S1", "(i,j->i,0,j)");
SetSTMap(prog, "S2", "(i,j->i,1,j)");
SetParallel(prog, "0", 2);
```

3.4.2 Memory Mapping

Memory allocation is specified through multi-dimensional affine functions with dimension-wise modulo operation, called modular mapping or pseudo projections. This representation is commonly used by existing approaches for optimal memory allocation [69, 90, 112, 115]. Another representation using integer lattices may also be represented as modular mappings [25].

For the 3-point Jacobi stencil example, memory allocation corresponding to the commonly used "ping-pong" style allocation is specified by the following script:

```
prog = ReadAlphabets("jacobi1D.ab");
SetMemoryMapping(prog, "A", "(t,i->t,i)", "2,0");
```

Two arrays of size `N` are used alternately in each iteration of the time loop, achieving the ping-pong style computation of Jacobi stencil. The convention is to treat modulo factor 0 as projection without modulo operations.

3.4.2.1 Reductions

AlphaZ provides two possible ways to specify schedules of reduction expressions. One is to view each reduction as "atomic" operations, and to schedule the domain of the answers of reductions. The other is to assign time stamps to each point in the body of reductions.

The latter provides more control over how the computations are executed, while the former is a simpler abstraction. Existing scheduling techniques for programs with reductions only handle the former case, and scheduling reduction bodies is an open problem, although a partial solution was proposed by Gautam et al. [41].

The latter option is only applicable if the reduce expression is the top-most expression of an equation. Then the space-time mapping given to the variable on the LHS of the equation can be specified to schedule the reduction body. In such cases, the LHS of the STmap must have the dimensionality equal to that of the reduction body.

Since reduction involves projection, the dimensionality is usually different between the reduction body and the answer space, and thus the same command `SetSTMap` may be used for both purposes.

3.4.3 Additional Specifications

Target Mapping described above forms the basis for specifying execution strategies orthogonal to the specification of *what* is computed. However, such a TMap is not yet complete. Additional, optional, specifications will be needed for code generators to accommodate other strategies such as tiling and synchronization, and possibly code generator specific options. The purpose of TMap is to decouple optimization strategies from the input specification and code generators, so that design space exploration and orthogonal specification are

both possible. Thus, it is best if options for future code generators are also decoupled and exposed through TMap.

3.4.3.1 Tiling

Tiling is a well known loop transformation for data locality and extracting coarse grained parallelism [47, 123]. The additional specification required for tiling is which dimensions are to be tiled, and tile sizes.

3.5 Code Generators

There are several code generators available in AlphaZ, and more are being developed. The code generator takes a program with Target Mapping and produces executable code. Current code generators all target C as the target output language.

How the Target Mapping is used largely depends on the code generator. Some specialized code generators will ignore certain specifications, or partially modify them to fit their purpose.

In this section, we describe two of the available code generators.

- WriteC: Demand driven code generator that does not require any TMap specification.
- ScheduledC: Code generator that fully respects the TMap specification. This code generator is intended to be the core for more specialized code generations.

The basic interface to AlphaZ generated code is function calls. The generated code includes a function for each system in `Alpha`, where all the inputs and parameters are given as function arguments. In the generated C code, we also take pointers to output arrays as function arguments.

3.5.1 WriteC

WriteC is designed to provide executable code for any legal input specification without specifying any additional information, in other words, not even a schedule is specified. This code generator does not need TMap to be given, indeed, if this code generator is called and a TMap is given, most of it will be ignored. However, it respects the memory allocation for input and output variables because this affects how the generated code interfaces with existing code. If the given program contains cyclic dependences, it is detected and flagged at run-time.

Demand driven code produced by this code generator traverses the dependences "backwards" from the outputs to find out the values required to compute the output. The order of evaluation is specified implicitly when a value is "demanded," hence the name demand-driven code generator.

However, a naive implementation of this execution strategy may introduce inefficiencies when a value is used multiple times. For example, the equation $\texttt{fib}(\texttt{n}) = \texttt{fib}(\texttt{n}-1) + \texttt{fib}(\texttt{n}-2)$, which define the Fibonacci

series , will require the result of `fib(n − 2)` when computing `fib(n)` and also when computing `fib(n − 1)`. To avoid such redundant computation, the demand-driven code uses memoization to keep track of previously computed values using the approach proposed by Wilde and Rajopadhye [122].

3.5.2 ScheduledC

ScheduledC produces loop programs that execute the computations in the order specified by the TMap. The STmap, memory mapping, and tiling specifications are all respected. The program produces C programs and uses OpenMP for parallelism.

CLooG [13] is used to generate loop nest that scans all equations in `Alpha`. The `Alpha` program is first normalized, and then each branch of the top-level case expression, becomes a separate statement in the generated loop program. Before generating statements and calling CLooG, the space-time mapping of each variable is reflected by applying Change of Basis (recall Section 2.1.12.2.) After the application of CoBs, lexicographic scan of the statement domains becomes the scanning order specified by the space-time mapping.

CLooG returns an Abstract Syntax Tree, called *clast* that represents the generated loops. If tiling is specified in the given TMap, we use D-Tiling [53] to transform the clast. The clast is then pretty printed as C loop nests and then combined with additional codes, such as statement bodies and memory allocations to produce the final output.

3.6 AlphaZ and Model-Driven Engineering

Model-Driven Engineering (MDE) [35] is a technique of software engineering that place models; abstractions of knowledge of the target domain; as the center of development. AlphaZ and GeCoS [48] are compiler infrastructures that use MDE techniques in their development [34]. These two compiler infrastructures share a common model for manipulating polyhedral objects.

We find MDEs to be an attractive technique for *research* compiler development because many challenges in compiler development are addressed by MDE tools. The modeling community has developed a number of tools to ease software development, and to derive as much as possible from the model specifications. These tools are often directly applicable to compiler development.

For example, data structures for model instances can be generated automatically from model specifications. There are now tools to generate parsers given concrete syntax, to transform models, and to pretty print a model instance.

The key connection between modeling and compiler is the observation that the compiler Intermediate Representations (IRs) are models. The compiler IR, which usually is an Abstract Syntax Tree, is an abstraction of the source program, and a compiler performs analyses and transformations on the AST. Then, all

tools for parser generation, model manipulation, and pretty printing can directly be used to build the main phases of a compiler.

We emphasize that MDE is attractive for *research* compilers, but not necessarily for production compilers. The production compilers need to be robust and efficient, whereas research compilers are only used for prototyping new ideas. When the performance of compiler itself is not the critical factor, overheads associated with generative programming in MDE are not major concerns.

AlphaZ use the Eclipse Modeling Framework for model driven development. We use four models in AlphaZ:

- `polyhedralIR`; the AST of Alpha programs. Most program transformations manipulate this model.
- `polyIRCG` is a separate model for code generation. This model captures high level structures of the desired code, such as functions and their layout, placement of memory allocations, main loops, and other code fragments. Such abstraction allows us to easily reuse common components across multiple code generators (e.g., malloc and free.)
- `polymodel`, a model shared between GeCoS and AlphaZ that abstracts polyhedral objects, such as domains and functions.
- `prdg` models the Polyhedral Reduced Dependence Graph, and are used for some of the analyses such as scheduling. This model is also shared with GeCoS.

These models are all developed using the Eclipse Modeling Framework (EMF) [1]. We briefly describe the three tools we use extensively in AlphaZ.

- Xtext [4] is a tool that we use for generating Alphabets parsers. It is more than a parser generator, and provides us with editors with syntax highlighting, content-assists, and many other features.
- Tom/Gom [2] is a term rewriting system that we use to manipulate our models. Some transformations are much easier to specify as re-writing compared to visitors. For example, normalization of Alpha programs is implemented using a set of rewriting rules, which is a much more clean and easy specification compared to visitor based implementations.
- Xtend [3] is a DSL for code generation. It encourages modularized design of code generators, and allows modules to be swapped seamlessly. For example, we have two implementations of memory allocators, one that allocates arrays of arrays for multidimensional arrays, and another that allocates a linearized array, and assigns appropriate pointers for multi-dimensional access. Most of the specifications for these variants are common, and thus shared. The specifications are concisely specified through template-based code generation of Xtend, where the majority of the output is specified with plain texts.

Perhaps one of the most powerful benefit of using MDE comes from the modeling activity itself, and generative programming. Research compilers are often developed by generations of students, working on different projects. We need students to share their code as much as possible, but the coding style can largely vary. Modeling helps in communication between students at a higher level, where implementation details are irrelevant.

Furthermore, the base data structures and code skeletons are generated from model specifications. This gives a homogenized code structure across all systems that use the same framework, and also matches the model specification. Such homogenization can manage code variations among students to some extent, making sharing of code less painful.

3.7 Summary and Discussion

We have presented a system for exploring analyses and transformations in the polyhedral model. The two key features in our system are:

- the ability to re-consider memory allocations, and
- explicit representation of reductions.

Polyhedral representations of programs are expressed as systems of equations; which can either be extracted from loop nests, or programmed directly in an equational language. These polyhedral programs are manipulated using script driven transformations, to reflect human analyses or domain specific knowledge to help guide optimizing translations. Then executable code is generated by specifying schedule, memory allocation, and other implementation details.

`AlphaZ` has a number of transformations and code generators, and others are actively being developed. In addition to what previous tools have focused on, we believe that exploring memory allocations is very important. We expect it to become even more important as we target distributed memory machines.

While many tools focus on fully automated program transformations, a tool like `AlphaZ` that expose as much control to the user is helpful in developing and prototyping new ideas.

Chapter 4

AlphaZ Case Studies

In this chapter, we illustrate possible uses of our system through two case studies that benefit from explicit representation of reductions and memory re-mapping.

Section 4.1 illustrates the importance of memory re-mapping, with a benchmark from `PolyBench/C 3.2` [84], and Section 4.2, presents an application of a very powerful transformation on reductions, called Simplifying Reductions. The work in Section 4.2 is an extension of the MS thesis of Pathan [82] and was done in collaboration with him and Gautam.

We describe the necessary elements of the Simplifying Reductions used in this case study in Section 4.2.2. Note that this is not a new contribution of this dissertation, and are re-illustration of the necessary subset for completeness sake. Please refer to the original article by Gupta and Rajopadhye [40] for the complete algorithm.

4.1 Case Study 1: Time-Tiling of ADI-like Computation

The Alternating Direction Implicit method is used to solve partial differential equations (PDEs). One of the stencil kernels in `PolyBench/C 3.2` [84], `adi/adi.c` resembles ADI computation.[1]

ADI with 2D discretization solves two sets of tridiagonal matrices at each time step. The idea behind ADI method is to split the finite difference system of equations of a 2D PDE into two sets: one for the x-direction and another for y. These are then solved separately, one after the other, hence the name *alternating direction implicit*.

Shown below is a code fragment from PolyBench, corresponding to the solution for one direction in ADI. When this code is given to PLuTo [16] for tiling and parallelization, PLuTo fails to find that all dimensions can be tiled, and instead, tiles the inner two loops individually. The key reason is as follows: the value written by `S0` is later used in `S3`, since computing `S3` at iteration `[t,i1,i2]` (written `S3[t,i1,i2]`) depends on the result of `S0[t,i1,i2]` and `S0[t,i1,i2-1]`. Since the dependence vector is in the negative orthant, this *value-based dependence* does not hinder tiling in any dimension.

```
for (t = 0; t < tsteps; t++) {
   for (i1 = 0; i1 < n; i1++)
```

[1] There is an error in the implementation, and time-tiling would not be legal for a correct implementation of ADI. The program in the benchmark nevertheless illustrates our point that existing tools are incapable of extract the best performance, largely because of lack of memory remapping.

```
     for (i2 = 1; i2 < n; i2++) {
S0:      X[i1][i2] = X[i1][i2] - X[i1][i2-1] * A[i1][i2]
                           / B[i1][i2-1];
S1:      B[i1][i2] = B[i1][i2] - A[i1][i2] * A[i1][i2]
                           / B[i1][i2-1];
     }

S2 ... // 1D loop updating X[*,n-1] (details irrelevant here)

     for (i1 = 0; i1 < n; i1++)
     for (i2 = n-1; i2 >= 1; i2--)
S3:      X[i1][i2] = (X[i1][i2] - X[i1][i2-1]
                * A[i1][i2-1]) / B[i1][i2-1];

   ... //second pass for i1 direction

  }
```

However, the original C code reuses the array `X` to store the result of `S0` as well as `S3`. This creates a memory-based dependence $\texttt{S3}[\texttt{t}, \texttt{i1}, \texttt{i2}] \rightarrow \texttt{S3}[\texttt{t}, \texttt{i1}, \texttt{i2}+1]$ because `S3[t,i1,i2]` overwrites `X[i1,i2]` used by `S3[t,i1,i2+1]`. Hence, `S3` must iterate in a reverse order to reuse array `X` as in the original code, whereas allocating another copy of `X` allows all three dimensions to be tiled.

4.1.1 Additional Complications

The memory-based dependences are the critical reason why the PLuTo scheduler (as implemented in Integer Set Library by Verdoolaege [121]) cannot find all three dimensions to be tilable in the above code. Moreover, two additional transformations are necessary to enable to scheduler to identify this. These transformations can be viewed as partially scheduling the polyhedral representation before invoking the scheduler. `AlphaZ` provides a command, called Change of Basis (CoB), to apply affine transforms to statements of polyhedral domains.[2]

One of them *embeds* `S2` which nominally has a 2D domain into 3D space, *aligning* it to be adjacent to a boundary of the domain of `S1`. The new domain of `S2` becomes $\{\texttt{t}, \texttt{i1}, \texttt{i2} \mid 0 \leq \texttt{t} < \texttt{tsteps} \wedge 0 \leq \texttt{i1} < \texttt{N} \wedge \texttt{i2} = \texttt{n} - 1\}$ (note the last equality).

The other complication is that a dependence from `S3` to `S2` is affine, not uniform ($\texttt{S3}[\texttt{t}, \texttt{i1}, \texttt{i2}] \rightarrow \texttt{S2}[\texttt{t}, \texttt{i1}, \texttt{n} - \texttt{i2} - 1]$) due to the reverse traversal of the `i2` loop `S3`. If a CoB ($\texttt{t}, \texttt{i1}, \texttt{i2} \rightarrow \texttt{t}, \texttt{i1}, \texttt{n} - \texttt{i2} - 1$) is applied to the domain of `S3` we get a uniform dependence. After these three transformations (removing memory-based dependences, and the two CoBs) the PLuTo scheduler discovers that all loops are fully permutable.

---

[2]This is similar to the preprocessing of code generation from unions of polyhedra [13], where affine transforms are applied such that the desired schedule is followed by lexicographic scan of unions of polyhedra. Since the program representation in `AlphaZ` is equational, any bijective affine transformation is a legal CoB.

We are not sure of the precise reason why PLuTo scheduling is not able to identify all dimensions are tilable without these transformations. Parts of PLuTo scheduling are driven by heuristics, and our conjecture is that these cases are not well handled. We expect these difficulties can be resolved, and that it is not an inherent limitation of PLuTo. However, a fully automated tool, prevents a smart user from so guiding the scheduler. We believe that guiding automated analyses can significantly help refining automated components of tools.

4.1.2 Performance of Time Tiled Code

Since PLuTo cannot tile the outer time loop, or fuse many of the loops due to the issues described above, PLuTo parallelized code contains 4 different parallel loops within a time step. On the other hand, `AlphaZ` generated code with time-tiling consists of a single parallel loop, executing wave-fronts of tiles in parallel. Because of this we expect the new code to perform significantly better.

We measured the performance of the transformed code on a workstation, and also on a node in Cray XT6m. The workstation uses two 4 core Xeon5450 processors (8 cores total), 16GB of memory, and running 64-bit Linux. A node in the Cray XT6m has two 12 core Opteron processors, and 32GB of memory. We used `GCC/4.6.3` with `-O3 -fopenmp` options on the Xeon workstation, and `CrayCC/5.04` with `-O3` option on the Cray. PLuTo was used with options `--tile --parallel --noprevector`, since prevector targets `ICC`.

`AlphaZ` was supplied with the original C code along with a script file specifying pre-scheduling transformations described above, and then used the PLuTo scheduler to complete the scheduling. Memory allocation was specified in the script as well, and additional copies of X were allocated to avoid the memory-based dependences discussed above. We used the `ScheduledC` code generator to produce the tiled and parallelized code.

For all generated programs, only a limited set of tile sizes were tried (8, 16, 32, 64 in all dimensions), and we report the best performance out of these. The problem size was selected to have cubic iteration space that runs for roughly 60 seconds with the original benchmark on Xeon environment ($\texttt{tsteps} = \texttt{n} = 1200$).

The results are summarized in Figure 4.1, confirming that the time-tiled version performs much better. On the Cray, we can observe diminishing returns of adding more cores with PLuTo parallelized codes, since only the inner two loops are parallelized. `AlphaZ` generated code does require more memory (this can actually, be further reduced), but at the same time, time-tiling exposes temporal reuse of the memory hierarchies.

4.2 Case Study 2: Complexity Reduction of RNA Folding

In this section, we show a detailed description of how a known optimization that reduce the complexity of RNA folding algorithm from $O(N^4)$ to $O(N^3)$ can be semi-automatically applied using a technique called Simplifying Reductions [40].

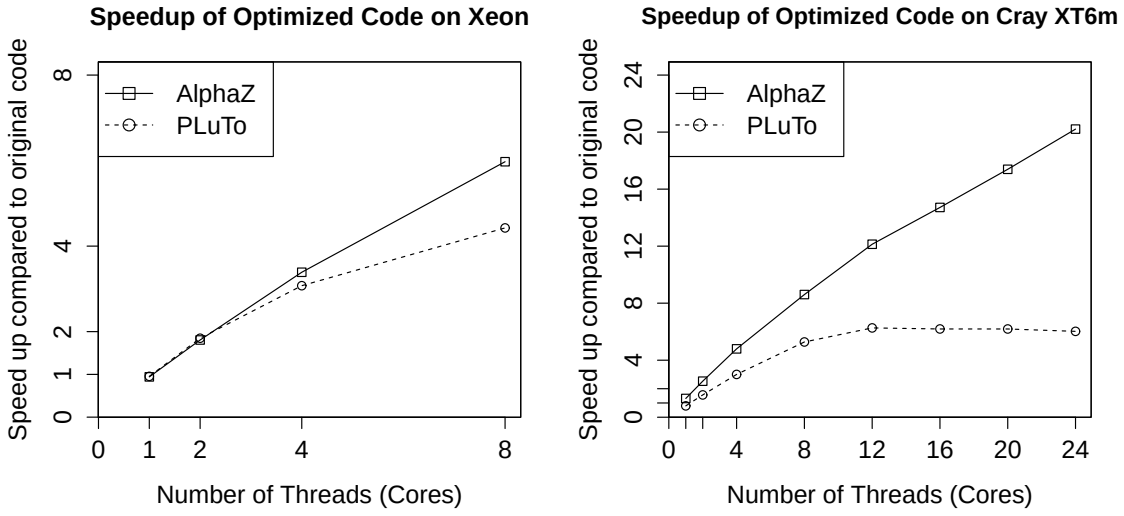


Figure 4.1: Speedup of `adi.c` parallelized with PLuTo and `AlphaZ`, with respect to the execution time of the unmodified `adi.c` from `PolyBench/C 3.2`. Observe that coarser grained parallelism with time-tiling leads to significantly better scalability with higher core count on the Cray.

RNA secondary structure prediction, or RNA folding, is a widely used algorithm in bio-informatics. The original algorithm has $O(N^4)$ complexity, but an $O(N^3)$ algorithm has been previously proposed by Lyngso et al. [73]. However, no implementation of the $O(N^3)$ algorithm has been made publicly available to the best of our knowledge.

The complexity reduction takes advantage of "hidden scans" in collections of reductions, where results (possibly partial) of a reduction can be reused in computing other reductions. For example, consider the following where X_i, $0 \leq i < N$ is computed as sums of subsets of values in $A_i; 0 \leq i < N$.

$$X_i = \sum_{k=0}^{i} A_k$$

This is actually a prefix (scan) computation, and can be written as the following:

$$X_i = \begin{cases} i = 0 : A_i \\ i > 0 : A_i + X_{i-1} \end{cases}$$

Note that the former equation takes $O(N^2)$ time while the latter takes $O(N)$ time. This is the core of the algorithm, and simplifying reductions consists of collection of analyses and transformations to detect and transform such reductions to corresponding scan computations.

A much more complicated simplification, but also based on reuse across reductions, was found by Lyngso et al. [73] in the RNA secondary structure prediction algorithm. However, implementing such optimization require significant re-structuring of the program. Moreover, although the Simplifying Reductions algorithm include the necessary analyses to identify hidden scans, detecting scans from a real application is non-trivial.

We present a systematic way of deriving reduced complexity implementation of a function in UNAfold software package [75], using the AlphaZ system. Although the Simplifying Reductions was proposed by Gupta and Rajopadhye [40], there is no known implementation of the algorithm aside from `AlphaZ`. This section is an extension to a related Master's Thesis by Pathan [82] that describe the systematic transformation of UNAfold in AlphaZ. We extend the work of Pathan by deriving the sequence of transformations to apply from the optimality algorithm presented by Gupta and Rajopadhye [40]. We also use a more efficient code generator and compare the performance of the optimized implementation with the original to empirically show the benefit of the optimization.

4.2.1 Intuition of Simplifying Reductions

We first illustrate the intuition using a simple example. The prefix sum computation can be expressed as the following:

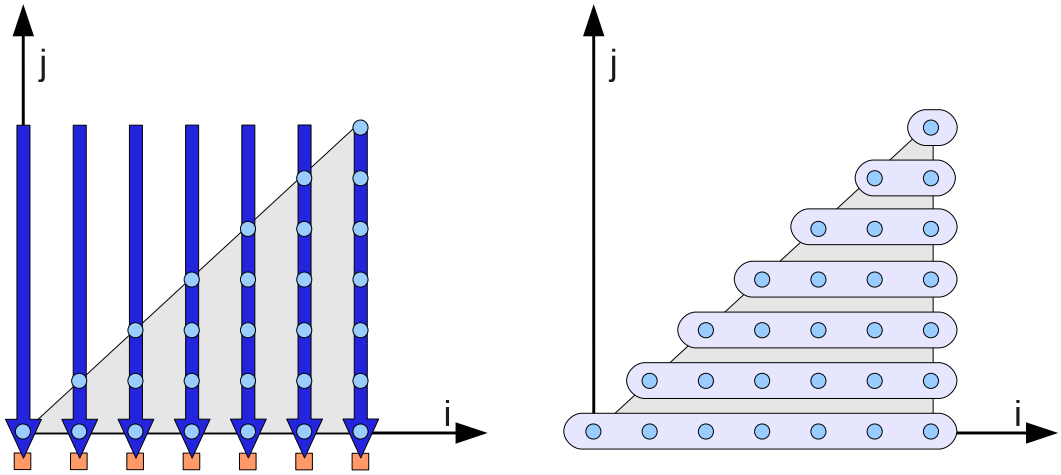
$$X[i] = \sum_{j=0}^{j=i} A[j] \tag{4.1}$$

with $\mathcal{D}_E = \{i, j | 0 \leq j \leq i < N\}$.

Figure 4.2 visualizes the iteration space of this program for N=8. The body of the reduction has a triangular domain $\{i, j | 0 \leq j \leq i < N\}$, and there are 7 independent reductions along the vertical axis. Because $\mathtt{A}[j]$ is accessed within a 2D domain, it can be observed that all points along the horizontal access that has the same j but different i all share the same value. Note that the reuse space, i.e., the set of points that share the same value, is spanned by the vector [1,0].

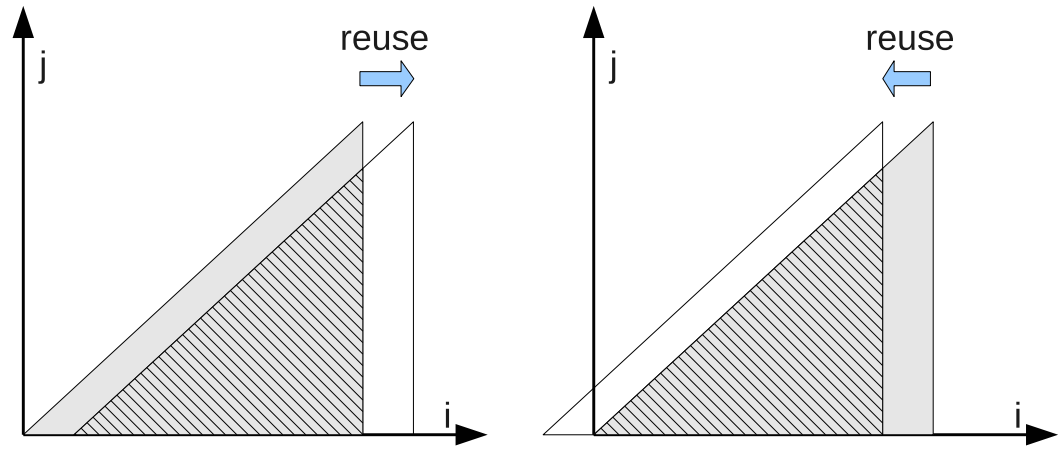
Assume that some constant vector in the reuse space, reuse vector r_E, is given as the input and the simplification is performed so that an instance of reduction at z reuses the result of another instance at $z - r_E$. Unless the values used at different instances of reductions are identical, reusing the result of another instance by itself is not enough. Because the iteration spaces are represented as polyhedra, the additional computation required can be computed. Figure 4.3 illustrates the reuse space and how the required computation in addition to the reuse is computed. Domain of additional computations are derived from the original domain $\mathcal{D}_E$ (filled domain) and its translation by the reuse vector $\mathcal{D}_{\mathtt{E}'}$ (unfilled domain). Domain with diagonal stripes is the intersection $\mathcal{D}_{int} = \mathcal{D}_E \bigcap \mathcal{D}_{\mathtt{E}'}$. $\mathcal{D}_{int}$ is where the result of two reductions r_E apart overlaps and can be reused. Thus, the diagonal strip of filled domain that does not have the stripe, $\mathcal{D}_{add} = \mathcal{D}_E - \mathcal{D}_{\mathtt{E}'}$ is the domain that needs to be computed in addition to the reuse.

Depending on the shape of the domain and the direction of reuse being exploited, some computation must be "undone" in addition to the reuse. In such cases, the reduction operator must have a corresponding inverse operator in order to undo parts of the computation. For example, if the vector[-1,0] was used instead in the above example, $P(x)$ is computed from $P(x+1)$ by *subtracting* $A[x+1]$. Such a domain, called *subtract domain*, can be computed as well, and it must be empty if the operator does not have an inverse.



(a) Iteration Space and Reductions (b) Reuse of `A`

Figure 4.2: Geometric illustration of the iteration space and reductions involved in prefix sum computation for N=8. The iteration space has a triangular domain where all integer points represent a computation. The reduction is along the vertical axis so that all points with the same i contributes to the same answer. Because `A` is indexed only with j, all points with the same j shares the same value.



(a) When reuse vector $(1, 0)$ is used (b) When reuse vector $(-1, 0)$ is used

Figure 4.3: Visualization of the reuse and simplification. $\mathcal{D}_{\mathtt{E}'}$ is the domain translated by the reuse vector. The intersection of the two domains (striped and filled) is the value being reused. In Figure (a), the diagonal strip of filled domain that does not have the stripe, $\mathcal{D}_{add} = \mathcal{D}_E - \mathcal{D}_{\mathtt{E}'}$ is the domain that needs to be computed in addition to the reuse. In Figure (b), the diagonal strip of unfilled domain, $\mathcal{D}_{sub} = \mathcal{D}_{\mathtt{E}'} - \mathcal{D}_E$ is the domain of values that needs to be undone from the reused value.

The core of Simplifying Reductions is in precisely computing these addition and subtraction domains through geometrical analysis, by shifting polyhedra along the reuse space.

4.2.2 Simplifying Reductions

We first introduce the notion of *share space* that characterize sharing of values, used to determine if a given r_E is legal or not. Then we introduce the simplification transformation, followed by other transformations that enhance the applicability of Simplifying Reductions.

4.2.2.1 Sharing of Values

Consider a dependence expression E of the form:

$$X.f \tag{4.2}$$

The expression E has the same value at any two index points $z, z' \in \mathcal{D}_E$ if $f(z) = f(z')$ since they map to the same index point of X. We will say that the value of E at these index points is *shared*. Note, two index points in $\mathcal{D}_E$ share a value if they differ by a vector in $\ker(f)$. Thus, values of E are shared along the linear space $\ker(f)$.

However, $\ker(f)$ may not be the maximal linear space along which values are shared in E. Observe, if in turn, index points in X also share values, along $\ker(f')$ say, then a larger linear space along which values of E are shared is $\ker(f' \circ f)$. We denote the maximal linear space along which values are shared in E as $\mathcal{S}_E$, and call it the *share space* of E. Below, we list the relationship between the share space of expressions. We assume that the specification has been transformed to have just input and computed variables, and all reductions are over expressions defined on a single integer polyhedron. The share space, S_E is equal to

- ϕ if E is a constant.
- ϕ is E is an input variable. We assume that program inputs have no sharing.
- $\mathcal{S}_X$ if E is a computed variable defined by $E = X$
- $\bigcap_{i=1}^{M} \mathcal{S}_{E_i}$ if E is $\mathrm{op}(E_1, \ldots, E_M)$
- $\bigcap_{i=1}^{M} \mathcal{S}_{E_i}$ if E is case $E_1, \ldots, E_M$ esac
- $\mathcal{S}_X$ if E is $\mathcal{D}' : X$
- $\ker(T \circ f)$ if E is $X.f$ and $\mathcal{S}_X = \ker(T)$.
- $f_p(\ker(Q) \cap \mathcal{S}_X)$ if E is reduce$(\oplus, f_p, X)$ and $\mathcal{D}_X$ is a single integer polyhedron $\mathcal{P} \equiv \{z | Qz + q \geq 0\}$.

4.2.2.2 Simplifying Transformation

The input reduction is required to be in the following form:

$$X = \text{reduce}(\oplus, f_p, E) \tag{4.3}$$

where $\mathcal{D}_E$ is a single integer polyhedron and equal to $\mathcal{X}_E$. For simplicity of explanation, we have the reduction named by a computed variable X.

The Simplifying Reduction transformation takes as inputs; a reduction in the form of Equation 4.3, where $\mathcal{D}_E$ is a single integer polyhedron, and a *legal* vector specifying the direction of reuse r_E; and returns a semantically equivalent equation:

$$\begin{aligned}
X &= \\
&\text{case} \\
(\mathcal{D}_{add} - \mathcal{D}_{int}) &: X_{add}; \\
(\mathcal{D}_{int} - (\mathcal{D}_{add} \cup \mathcal{D}_{sub})) &: X.(z \to z - r_X); \\
(\mathcal{D}_{add} \cap (\mathcal{D}_{int} - \mathcal{D}_{sub})) &: (X_{add} \oplus X.(z \to z - r_X)); \\
(\mathcal{D}_{sub} \cap (\mathcal{D}_{int} - \mathcal{D}_{add})) &: (X.(z \to z - r_X) \ominus X_{sub}); \\
(\mathcal{D}_{add} \cap \mathcal{D}_{int} \cap \mathcal{D}_{sub}) &: (X_{add} \oplus X.(z \to z - r_X) \ominus X_{sub}); \\
&\text{esac}; \\
X_{add} &= \text{reduce}(\oplus, f_p, (\mathcal{X}_E - \mathcal{X}_{E'}) : E) \\
X_{sub} &= \text{reduce}(\oplus, f_p, \\
&\qquad f_p^{-1}(\mathcal{D}_{int}) : (\mathcal{X}_{E'} - \mathcal{X}_E) : E')
\end{aligned}$$

where $E' = E.(z \to z - r_E)$, $r_X = f_p(r_E)$, $\ominus$ is the inverse of $\oplus$, $\mathcal{D}_{add}$, $\mathcal{D}_{sub}$ and $\mathcal{D}_{int}$ denote the domains $f_p(\mathcal{X}_E - \mathcal{X}_{E'})$, $f_p(\mathcal{X}_{E'} - \mathcal{X}_E)$ and $f_p(\mathcal{X}_E \cap \mathcal{X}_{E'})$ respectively, and X_{add} and X_{sub} are defined over the domains $\mathcal{D}_{add}$ and $\mathcal{D}_{int} \cap \mathcal{D}_{sub}$ respectively.

We require that the reuse vector r_E to satisfy $r_E \in S_E \setminus \ker(f_p)$ for the semantic to be preserved. Since r_E is the direction of reuse it must be in the share space. However, it must not be in the kernel of the projection function f_p. This is because the transformation involves the use of the value of X at an index point to simplify the computation at another and so in order to avoid a self-dependence, we must ensure that these index points are distinct (i.e., $r_X = f_p(r_E) \neq 0$).

Note that the above transformation requires the inverse operator $\ominus$, which may not exist for some $\oplus$. Then all branch of the case in transformed that use $\ominus$ must have empty context domains.

4.2.2.3 Simplification Enhancing Transformations

We have shown a transformation that resulted in the simplification of reductions. Here, we will present transformations that, *per se*, do not simplify but enhance simplification. The goal of enhancing transformations is to increase the applicability of simplification by enlarging S_E. We only present a subset of such transformations used in simplification of RNA folding we show in Section 4.2.5.

4.2.2.4 Distributivity

Consider a reduction of the form

$$E = \text{reduce}(\oplus, f_p, E_1 \otimes E_2)$$

where $\otimes$ distributes over $\oplus$.

If one of the expressions is constant within the reduction (E_1, say), we would be able to distribute it outside the reduction. For the expression E_1 to be constant within a reduction by the projection f_p, we require

$$\mathcal{H}_{\mathcal{D}_E} \cap \ker(f_p) \subseteq \mathcal{H}_{\mathcal{D}_E} \cap \mathcal{S}_{E_1}$$

where $\mathcal{H}_{\mathcal{D}}$ is defined as the linear part of the smallest affine subspace containing $\mathcal{H}_{\mathcal{D}}$. $\mathcal{H}_{\mathcal{D}}$ becomes important when the domains contain equalities. After distribution, the resultant expression is

$$E_1 \otimes \text{reduce}(\oplus, f_p, E_2)$$

The resultant expression can potentially have larger share space, since share space of E_1 no longer affects that of the reduction body.

4.2.2.5 Reduction Decomposition

We will now introduce a transformation that has wide applicability in enhancing simplification.

An expression of the form

$$\text{reduce}(\oplus, f_p, E)$$

is semantically equivalent to

$$\text{reduce}(\oplus, f''_p, \text{reduce}(\oplus, f'_p, E))$$

where $f_p = f''_p \circ f'_p$.

This transformation enhances simplification primarily by exposing additional opportunities to apply distributivity. When a reduction from Z^n to Z^m where m is at least 2 dimensions less than n, then some expression that cannot be distributed may be distributed once the reduction is decomposed.

For example, consider the following reduction:

$$\text{reduce}(\oplus, (i, j, k \to i), E_1 \otimes E_2)$$

where $S_{E_1} = \phi$, $S_{E_2} = [0,0,1]$, and $\ker(f_p) = ([0,1,0],[0,0,1])$. Since $\ker(f_p) \not\subseteq S_{E_2}$, E_2 cannot be distributed out. However, applying reduction decomposition with $f'_p = (i,j,k \to i,j)$ and $f''_p = (i,j \to i)$ to obtain:

$$X = \text{reduce}(\oplus, (i,j,k \to i,j), E_1 \otimes E_2)$$
$$\text{reduce}(\oplus, (i,j \to i), X)$$

allows E_2 to be distributed out from the inner reduction.

Depending on its use, the reduction decomposition may or may not have side effects. However, the case without side effects only occur when domains of reduction body contain equalities (or some constant "thickness" variations of equalities) along certain dimensions. When there are equalities in the domain of reductions, the space spanned by the equalities are separated by reduction decomposition as a pre-processing. These cases, including constant "thickness" variants, are formalized as *Effective Linear Subspace* in the original article [40]. It states that a polyhedron $\mathcal{P}$ have constant thickness along any vector *not* in its effective linear subspace $\mathcal{L}_{\mathcal{P}}$.

For the domains in UNAfold, $\mathcal{L}_{\mathcal{D}_E}$ is the universe, and we focus on reduction decomposition with side effects. Reduction decomposition with side effects reduce the space of possible reuse directions, and affects if simplification is applicable later in the sequence of transformations.

Recall that distributing an expression E out from the reduction with a projection f_p requires $\ker(f_p) \subseteq \mathcal{S}_E$. Therefore, we may decompose f_p into $f''_p \circ f'_p$ to distribute an expression with available reuse space S_E outside the inner reduction by choosing f'_p such that

$$\ker(f'_p) = \ker(f_p) \cap \mathcal{S}_E$$

4.2.3 Normalizations

There are a number of transformations for taking equations with reductions into the form required by the simplification transformation (Equation 4.3). We introduce two of such transformations that are used later in Section 4.2.5.

4.2.3.1 Normalize Reductions

Normalize Reductions is a transformation that takes expression containing reductions:

$$E = \cdots \text{reduce}(\oplus, f_p, E_1) \cdots$$

and isolates reductions by adding vairables:

$$E = \cdots X \cdots$$
$$X = \text{reduce}(\oplus, f_p, E_1)$$

After this transformation, all reduce expression in the Alphabets program will be top-level expressions (the first expression in the right hand side of an equation). This is purely a pre-processing to obtain reductions of the form required by the simplification algorithm. We also provide another transformation, called Inline, to replace variables with its definition, so that the variables introduced by this transformation can eventually be removed.

4.2.3.2 Permutation Case Reduce

Permutation Case Reduce, presented as a theorem by Le Verge [65], takes reduce expression of the form:

$$E = \text{reduce}(\oplus, f_p, \text{case } E_1; E_2; \text{ esac})$$

and returns a semantically equivalent equation:

$$\begin{aligned} E = \text{case} \\ \mathcal{D}_1 &: X_1; \\ \mathcal{D}_{12} &: (X_1 \oplus X_2); \\ \mathcal{D}_2 &: X_2; \\ \text{esac}; \end{aligned}$$

where $\mathcal{D}_{12} = f_p(\mathcal{D}_{E_1}) \cap f_p(\mathcal{D}_{E_2})$, $\mathcal{D}_1 = f_p(\mathcal{D}_{E_1}) \setminus f_p(\mathcal{D}_{E_2})$, $\mathcal{D}_2 = f_p(\mathcal{D}_{E_2}) \setminus f_p(\mathcal{D}_{E_1})$, and X_1, X_2 are defined as follows:

$$X_1 = \text{reduce}(\oplus, f_p, E_1)$$
$$X_2 = \text{reduce}(\oplus, f_p, E_2)$$

The transformation essentially moves case expressions out of the reduction. Since the simplification transformation requires that the domain of the reduction body to be a single polyhedron, and not unions of polyhedra, case expressions must be moved out.

4.2.4 Optimality and Algorithm

The optimality algorithm proposed by Gupta and Rajopadhye [40] considers a number of transformations that can expose more simplification opportunities. In this dissertation, we outline a simplified version of the algorithm used when applying the algorithm to UNAfold shown in Algorithm 4.1.

For the dynamic programming in Step 5 to work, we must show that from the infinite search space of parameters for the various transformations, we need to consider only a finite set of choices and the global

Algorithm 4.1 The Simplification Algorithm: Subset for UNAfold

Input:

An equational specification in the polyhedral model.

Output:

An equivalent specification of optimal complexity.

Algorithm:

1. Preprocess to obtain a reduction over an expression whose domain is a single polyhedron and equal to its context domain.
2. Other pre-processing not used for UNAfold, if applicable.
3. Perform any of the following transformations, if applicable.
   (a) Distributivity.
   (b) Other side-effect free enhancing transformations not used for UNAfold.
4. Repeat Steps 1-3 till convergence.
5. Dynamic Programming Algorithm to optimally choose:
   (a) The simplification transformation along some r_E.
   (b) A reduction decomposition with side effects.
   (c) Other enhancing transformations with side-effects, not used for UNAfold.
6. Repeat from Step 1 on residual reductions until convergence.

optima can be reached through such choices. The intuition behind our formal argument used by Gupta and Rajopadhye [40] is as follows:

Simplification Transformation: From the infinitely many choices of reuse vectors r_E in a share space, we show that there are only finitely many equivalence classes. The intuition is that the result of applying the simplification transformation with different reuse vectors from the same equivalence class are identical except for the "thickness" of the residual reductions. The residual reductions corresponds to the addition and subtraction domains ($\mathcal{D}_{add}, \mathcal{D}_{sub}$). The constant "thickness" of reductions affect only the constant factors, but not the asymptotic complexity.

Reduction Decomposition with Side Effects for Distributivity: Of the infinite possible decompositions of the projection f_p, we only need to consider a finite subset, since the transformation is needed to distribute a set of subexpressions outside the inner reduction. The number of candidates are finite, with the equivalence classes formed by the basis vectors of its kernels.

4.2.5 Application to UNAfold

Finally, we show that the application of the simplification algorithm described in Section 4.2.4 leads to the reduced complexity algorithm.

The RNA folding algorithm is a dynamic programming algorithm. There are multiple variations of the algorithm based on the cost model used. UNAfold [75] uses a prediction model based on thermodynamics that finds a structure with minimal free energy. For an RNA sequence of length N, the algorithm computes multiple tables of free energy for each subsequence from i to j such that $1 \leq i \leq j \leq N$. The three tables $Q(i,j)$, $Q'(i,j)$, and $QM(i,j)$ correspond to the free energy for three different substructures that may be formed.

The following equations taken from the original algorithm:

$$Q(i,j) = \min \left\{ \begin{array}{l} b + Q(i+1,j), \\ b + Q(i,j-1), \\ c + E_{ND}(i,j) + Q'(i,j), \\ b + c + E_{5'D}(i+1,j) + Q'(i+1,j), \\ b + c + E_{3'D}(i,j-1) + Q'(i,j-1), \\ 2b + c + E_{DD}(i+1,j-1) + Q'(i+1,j-1), \\ QM(i,j) \end{array} \right\} \tag{4.4}$$

$$Q'(i,j) = \min \left\{ \begin{array}{l} E_H(i,j), \\ E_S(i,j) + Q'(i+1,j-1), \\ \min_{i<i'<j'<j} \left\{ E_{BI}(i,j,i',j'), Q'(i',j') \right\}, \\ a + c + E_{ND}(j,i) + QM(i+1,j-1), \\ a + b + c + E_{3'D}(j,i) + QM(i+2,j-1), \\ a + b + c + E_{5'D}(j,i) + QM(i+1,j-2), \\ a + 2b + c + E_{DD}(j,i) + QM(i+2,j-2) \end{array} \right\} \tag{4.5}$$

$$QM(i,j) = \min_{i+1 \leq k \leq j-2} Q(i,k-1) + Q(k,j) \tag{4.6}$$

where a,b, and c, are constants and functions of the form E_{XY} are all energy functions for different substructures.

The third term in Equation 4.5 is the dominating term that makes the algorithm $O(N^4)$. Notice that the term uses four free variables i,j,i' and j' and has a four dimensional domain $\{i,j,i',j' | 1 \leq i < i' < j' < j \leq N\}$ and hence $O(N^4)$ complexity. The term corresponds to a substructure called internal loops, and the cubic time algorithm to evaluate this term is referred to as *fast i-loop*.

4.2.5.1 Simplification

We focus on the dominating term in calculating the energy associated with internal loops to illustrate the simplification. The term is rewritten as a separate equation using our notation of reductions, and named QBI (since it is the term that involves E_{BI}) is the following:

$$QBI[i,j] = \texttt{reduce}(\min, (i,j,i',j' \rightarrow i,j), E_{BI}(i,j,i',j') + Q'(i',j')) \tag{4.7}$$

The sequence of transformations to obtain the above corresponds to Step 1 in Algorithm 4.1.

Before simplification, the energy function E_{BI} must be inlined to expose the reuse. This inlining is *not* part of the algorithm, and requires human analysis to deduce that the inlining is necessary at the moment.

E_{BI} has two different definitions, one for the generic case and another to handle special cases. These special cases are when the size of the internal loop is very small (less than 4) and thus resembles other kind of substructures. Since the special case can be described as polyhedral domains, we focus on the generic case for simplicity.

The function E_{BI} for generic case is defined as follows:

$$E_{BI}(i,j,i',j') = Asym(i'-i-j+j') + S_P(i'-i+j-j'-2) + E_S(i,j) + E_S(i',j') \tag{4.8}$$

Inlining Equation 4.8 into Equation 4.7 gives the following:

$$QBI[i,j] = \texttt{reduce}\left(\min, (i,j,i',j' \rightarrow i,j), \begin{cases} Asym(i'-i-j+j') & + \\ S_P(i'-i+j-j'-2) & + \\ E_S(i,j) & + \\ E_S(i',j') & + \\ Q'(i',j') & \end{cases}\right) \tag{4.9}$$

Computing the share space for each sub-expressions in the reduction body (recall Section 4.2.2.1) gives:

$$
\begin{aligned}
S_{Asym}(i' - i - j + j') =& [-1, 1, 0, 0], [1, 0, 1, 0], [1, 0, 0, 1] \\
S_{S_P}(i' - i + j - j' - 2) =& [1, 0, 1, 0], [1, 1, 0, 0], [-1, 0, 0, 1] \\
S_{E_S}(i, j) =& [0, 0, 1, 0], [0, 0, 0, 1] \\
S_{E_S}(i', j') =& [1, 0, 0, 0], [0, 1, 0, 0] \\
S_{Q'}(i', j') =& [1, 0, 0, 0], [0, 1, 0, 0]
\end{aligned}
$$

In addition, the kernel of the projection function $\ker(f_p) = ([0, 0, 1, 0], [0, 0, 0, 1])$. Since the share space of $E_S(i, j)$ contains $\ker(f_p)$, it can be distributed out from the reduction in Step 3a of Algorithm 4.1 to produce: [3]

$$
Q_{BI}[i, j] = E_S(i, j) + \texttt{reduce}\left(\min, (i, j, i', j' \to i, j), \begin{cases} Asym(i' - i - j + j') & + \\ S_P(i' - i + j - j' - 2) & + \\ E_S(i', j') & + \\ Q'(i', j') \end{cases} \right) \tag{4.10}
$$

Taking the intersection of share spaces of the remaining terms gives the zero vector, and therefore no reuse can be exploited. This takes us to Step 5b of Algorithm 4.1. We analyze the share space of expressions in the reduction body and the projection function to find candidate decompositions. The candidate f'_p are:[4]

$$
\begin{aligned}
\ker(f'_p) &= \ker(f_p) \cap S_{Asym}(i' - i - j + j') \\
\ker(f'_p) &= \ker(f_p) \cap S_{S_P}(i' - i + j - j' - 2) \\
\ker(f'_p) &= \ker(f_p) \cap S_{E_S}(i', j') \\
\ker(f'_p) &= \ker(f_p) \cap S_{Q'}(i', j')
\end{aligned}
$$

The latter two candidates have no feasible f'_p because the intersection of $\ker(f_p)$ and its respective share space is empty. Similarly, $S_{Asym}(i' - i - j + j')$ do not have any feasible f'_p. The only feasible candidate is $\ker(f'_p) = [0, 0, 1, 1]$ with $S_{S_P}(i' - i + j - j' - 2)$.

The function $(i, j, i', j' \to i, j, j' - i')$ is a function that have the space spanned by $[0, 0, 1, 1]$ as its kernel. We use this function as f'_p with its corresponding $f''_p = (i, j, d \to i, j)$. Note that $f''_p \circ f'_p = (i, j, i', j' \to i, j)$ is the original projection. The choice of the function among the set of functions that have the same kernel

[3] $\mathcal{H}_{\mathcal{D}_E}$ is universe for this program.

[4] Note that once we have f_p and f'_p, f''_p can be deduced with standard linear algebra.

does not affect the resulting complexity. Decomposing the reduction with f'_p and naming the inner reduction QBI' gives the following two equations.

$$QBI[i,j] = E_S(i,j) + \texttt{reduce}(\min, (i,j,d \to i,j), QBI'[i,j,d]);$$
$$QBI'[i,j,d] = \texttt{reduce}\left(\min, (i,j,i',j' \to i,j,j'-i'), \begin{cases} Asym(i'-i-j+j') & + \\ S_P(i'-i+j-j'-2) & + \\ E_S(i',j') & + \\ Q'(i',j') & \end{cases}\right)$$

After the decomposition, the expression $S_P(i'-i+j-j')$ can be distributed out from the inner reduction, because its share space ($[1,1,0,0], [1,0,1,0], [-1,0,0,1]$) contains $\ker(f'_p) = [0,0,1,1]$ ($[1,0,1,0]+[-1,0,0,1] = [0,0,1,1]$.)

$$QBI'[i,j,d] = \texttt{reduce}\left(\min, (i,j,i',j' \to i,j,j'-i'), \begin{cases} Asym(i'-i-j+j') & + \\ E_S(i',j') & + \\ Q'(i',j') & \end{cases}\right)$$
$$+ S_P(-i+j-d-2)$$

Then the remaining expressions have a common share space spanned by the vector [-1,1,0,0]. Applying the simplifying reduction transformation using [-1,1,0,0] as the reuse vector yields an equivalent equation of the following form:

$$QBI'[i,j,d] = \begin{cases} \mathcal{D}_{init} & : X_{add} \\ \mathcal{D}_{add} & : \min(X_{add}, QBI'[i+1,j-1,d]) \end{cases} \tag{4.11}$$
$$+ S_P(-i+j-d-2)$$
$$X_{add} = \begin{cases} Asym(i'-i-j+j')+ \\ E_S(i',j')+ \\ Q'(i',j') \end{cases}$$

Domains D_{init}, and D_{add} are computed following the definitions in Section 4.2.2. The full Alphabets program after transformation in the appendix show the domains as the domain of corresponding Alpha variables.

In the following, we show a fragment of the Alphabets after sequence of transformations described above has been applied. We can observe that equations `QBI_SR1_init` and `QBI_SR1_add` both have equalities in the restrict expression. These equations respectively correspond to branches of QBI' in Equation 4.11. Because of the equalities the context domains of these reductions are 3D domains embedded in 4D space. Hence, we confirm that the complexity is reduced to $O(N^3)$. We also note that the term S_P was factored out since the simplification algorithm requires reduce expression to be the top-level expression.

```
//Simplifying Reduction result
QBI_SR1[i,j,ip] = case
   {|i-j+ip+7>= 0} : QBI_SR1_init;  //D_init case
   {|j-1-ip-8>= 0} : (QBI_SR1_add min QBI_SR1[i+1,j-1,ip]); //D_add case
esac;
```

```
//X_add for D_init
QBI_SR1_init[i,j,ip] = reduce(min, (i,j,ip,jp->i,j,jp-ip),
   {|ip-i==2} || {|j-jp==4 && ip-i==3} :
      ((Ebi_stacking([jp],[ip]) + Ebi_asymmetry([ip-i-1],[j-jp-1]))
                                + Qprime[ip,jp])
);

//X_add for D_add
QBI_SR1_add[i,j,ip] = reduce(min, (i,j,ip,jp->i,j,jp-ip),
   {|ip-i==2} || {|j-jp==4} :
      ((Ebi_stacking([jp],[ip]) + Ebi_asymmetry([ip-i-1],[j-jp-1]))
                                + Qprime[ip,jp])
);
```

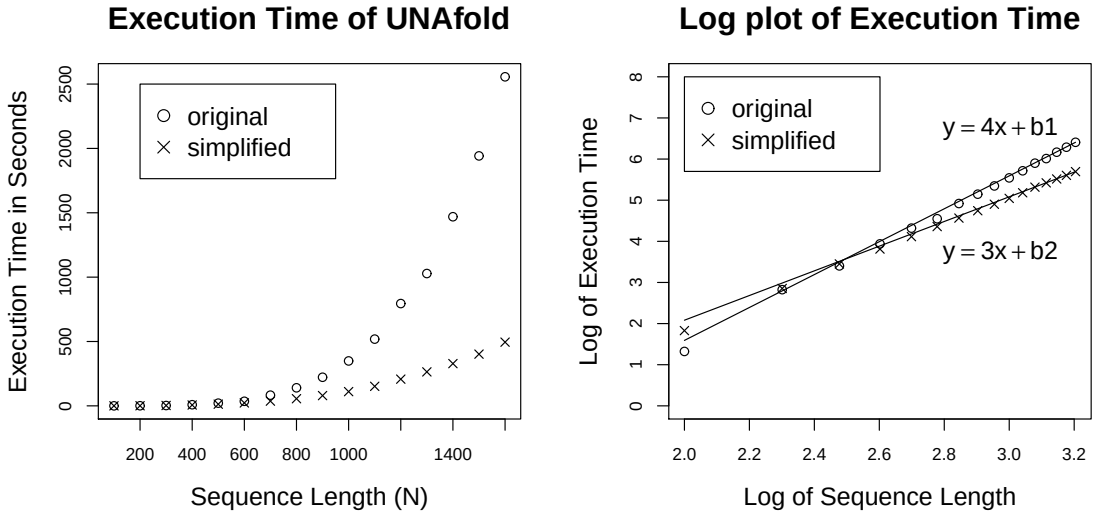
Starting from the original equations expressed as `Alpha`, the only step that is not handled by the algorithm is inlining to reach Equation 4.9 from Equation 4.7.

4.2.6 Validation

We have applied the above transformation using AlphaZ to the UNAfold 3.8 [75]. The function `fillMatrices_1` in `hybrid-ss-min.c` was written in our equational language, and the simplifying transformation was applied. The `ScheduledC` code generator was used to generate the simplified version of `fillMatrices_1` and replaced with the original function.

Both original and the simplified versions were compiled with `GCC/4.5.1`, with `-O3` option and the execution times were measured a machine with Core2Duo 1.86GHz and 6GB of memory running Linux. Because the default option of UNAfold limits the internal loop size to 30, we also set the limit to infinity when running `hybrid-ss-min`.

Figure 4.4 shows the measured performance, and its log-log scaled version. The log-scale plot clearly shows the reduction in complexity, and, as expected, the speedups with transformed code becomes greater and greater as the sequence length grows.



(a) Raw Execution Time

(b) Log-Log Plot

Figure 4.4: Execution Time of UNAfold after simplifying reduction compared with the original implementation. The two lines show are with slopes 4 and 3 with constant offsets (b1, b2) to make the lines meet the points at log(N)=3.2.

Chapter 5

"Uniform-ness" of Affine Control Programs

In this chapter, we discuss the "uniform-ness" of affine control programs. The polyhedral model can handle affine dependences and most techniques for multi-core processors are formalized for programs with affine dependences. If the target architecture has shared memory, affine accesses to shared memory have little or no difference in cost compared to uniform accesses. However, affine dependences are significantly more complex, and as soon as we reach beyond shared memory, handling affine dependences again becomes a problem. When generating distributed memory parallel code, affine dependences can result in broadcast or broadcast to complicated subsets of the processors. The pattern of communication can be affected by the problem size and by tile sizes if tiling is used.

When parallelization was a small niche; before the rise of multi-cores, a class of techniques called uniformization or localization, which replaces affine dependences with uniform dependences was of great interest [20, 74, 95, 104, 117, 128]. There is also a related approach for converting affine dependences to uniform dependences that "fold" the iteration space using piece-wise affine transformations [81, 129].

When the target architecture is hardware, such as FPGAs, VLSI circuits, or ASICs, implementing communications induced by affine dependences are very expensive. For example, communications among 1D processors over $\{i|0 \leq i \leq 10\}$, that corresponds to an affine dependence $(i \rightarrow 0)$ require all processors to be connected to processor 0. Instead, if we replace the affine dependence with a uniform dependence $(i \rightarrow i-1)$, and propagate the value, only neighboring processors need to be connected. The process of replacing an affine dependence with uniform dependences is called uniformization or localization, and has a significant impact on the performance of hardware, especially on the die area consumed by implementing communications. Since shared memory hides these issues from the software layer, techniques for uniformization are rarely used in recent work toward automatic parallelization.

In this chapter, we ask ourselves the question: "How affine are affine control programs?", and re-evaluate the affine-ness of affine dependences in realistic applications. We use `PolyBench/C 3.2` as our benchmark, which consists of 30 benchmarks, and show that most of `PolyBench` can be easily uniformized.

5.1 Uniformization by Pipelining

There are a number of techniques for uniformization [20, 74, 95, 104, 117, 128]. In this section, we illustrate one of the techniques called uniformization by pipelining. This is one of the simpler methods for uniformization that replaces an affine dependence with chains of uniform dependences such that the same value eventually is propagated. It turns out that this method alone can uniformize most of `PolyBench`.

Let us first assume that the domain of all statements in a program has been transformed such that all statements have the same number of dimensions. This means that all dependences are now from $\mathbb{Z}^N$ to $\mathbb{Z}^N$, which also means that the linear part of the dependence, A, is an $N \times N$ square matrix. This is obviously a necessary condition of dependences to be affine, and is the starting point of all known uniformization techniques. The necessary pre-processing to take programs into this form, called embedding or alignment, is discussed later in Section 5.3.

Pipelining is only applicable to affine dependences that are idempotent [104]. An affine dependence is said to be idempotent if the linear part is idempotent. In other words, let $f = Ax + b$, then f is idempotent iff $A = A.A$. We describe nullspace pipelining below, based on the work by Roychowdhury [104] with some of the terminology replaced by those used in the polyhedral model [117].

Given an affine dependence with function $f = Ax + b$ and domain $\mathcal{D}$ is an N dimensional space, we first find an N-vector, called the uniformization vector v that satisfied the following:

- $v \in \ker(f)$; in the kernel (nullspace) of f, and
- $v \in \mathcal{H}_{\mathcal{D}}$; in the linearity space of $\mathcal{D}$

The kernel of a dependence characterizes the share space, i.e., the set of points that depend on the same value. The pipelining must be along those set of points, otherwise the value propagated cannot be used. The second condition is to prevent cases where there is only one point in the intersection of the space spanned by v, and $\mathcal{D}$, because the vector is not aligned with the equalities.

Once a vector v is chosen, we compute the following domains:

- $\mathcal{D}^0$: $\{x | x \in \mathcal{D} \wedge f(x) = x\}$; the initialization domain,
- $\mathcal{D}^+$: $\{x | x \in \mathcal{D} \wedge y \in \mathcal{D}^0 \wedge x = y + \alpha v \wedge \alpha > 0\}$; the positive propagation domain, and
- $\mathcal{D}^-$: $\{x | x \in \mathcal{D} \wedge y \in \mathcal{D}^0 \wedge x = y - \alpha v \wedge \alpha > 0\}$; the negative propagation domain.

The intuition is that the initialization domain is set of points where the dependence function is equivalent to identity. This set of points initializes the pipelining. The propagation domains are the points pipelining along v can reach, separated into two based on the direction of propagation.

Finally the equation of the following form:

$$X(z) = \cdots Y(f(z)) \cdots$$

is transformed as follows:

$$X(z) = \cdots \begin{cases} \mathcal{D}^0 : Y(z) \\ \mathcal{D}^+ : Y^+(z) \\ \mathcal{D}^- : Y^-(z) \end{cases} \cdots$$

$$Y^+(z) = \begin{cases} Y^+(z - v) \\ Y(z) \end{cases}$$

$$Y^-(z) = \begin{cases} Y^-(z + v) \\ Y(z) \end{cases}$$

where Y is the variable being accessed with an affine dependence, Y^+ and Y^- are variables introduced during the uniformization, called *uniformization variables*. In practice, some of the domains may be empty, making Y^+ or Y^- unnecessary.

Roychowdhury [104] showed that the above transformation is always applicable if the dependence function is idempotent, and the row rank of $I - A$ is 1. If the rank r is greater than 1, the dependence must be decomposed and uniformized r times.

5.2 Uniform in Context

An important analysis, especially for polyhedral representations extracted from loop programs, is to check if any "uniform-ness" is hidden by equalities. For example, a dependence with affine function $(i \to 0)$, with domain $\{i | i = 1\}$, is actually uniform, since the function can be replaced with $(i \to i - 1)$. These dependences are said to be uniform in *context*, where the dependences are uniform only for a certain domain [63, 74].

Uniform in context can be performed using Gauss-Jordan elimination. Given a dependence with function $f = Ax + b$ for some domain $\mathcal{D}$ on a N dimensional space. We first find the equalities in domain $\mathcal{D}$, denoted E. Then we apply Gauss-Jordan elimination on matrix $\begin{bmatrix} A \\ E \end{bmatrix}$ to obtain its reduced row echelon form, R. If the top $N \times N$ sub-matrix of R is the identity, then the function f is uniform in the context of $\mathcal{D}$ (or more precisely, E). Gauss-Jordan elimination only uses elementary row operations, and if the identity can be reached by some combination of elementary row operations, it is some linear combination of the rows (and hence is affine). The combination, or the necessary transformation, to make the dependence uniform may be deduced by keeping track of the elementary row operations applied.

In addition, even if the dependence is not completely uniform in context, the same procedure may be applied to *partially* uniformize a dependence in context. In reduced row echelon form, even if the $N \times N$ sub-

matrix is not the identity, some smaller sub-matrix (possibly 0×0) will be identity, and for the corresponding dimensions, the dependence is uniformized.

5.3 Embedding

One important pre-processing step, called *embedding* or alignment, is a step that transforms all statement domains to have the same number of dimensions. Since two statements must be in the same dimension for a dependence to even have a chance to be uniform, this pre-processing is essential.

Embedding of a statement is performed by applying change of basis with an affine function f from $\mathbb{Z}^n$ to $\mathbb{Z}^m$ where $n < m$. Any embedding is legal provided the function f is bijective. However, finding a good f is a very difficult problem, because the choice of f can have a huge influence on even whether or not a dependence can be uniformized with pipelining and/or if a dependence is already uniform after embedding.

Consider the following equation X with $\mathcal{D}_X = \{i | 0 \leq i \leq N\}$ that depends on Y with affine function $(i \to i, 0)$:

$$X(i) = Y(i, 0)$$

Applying CoB to X with function $(i \to i, 0)$ results in a uniform dependence, after uniformizing in context. However, if we instead apply CoB with $(i \to 0, i)$, the result is the following:

$$X(i, j) = Y(j, 0)$$

where $\mathcal{D}_X = \{i, j | 0 \leq j \leq N \wedge i = 0\}$ after CoB.

The linear part A of the dependence $(i, j \to j, 0)$ is $\begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}$, and is not idempotent. Therefore, the resultant dependence after CoB cannot be uniformized with pipelining. In fact, it requires adding a dimension, and we can say that embedding with $(i \to i, 0)$ is better.

The problem of automatically determining legal (i.e., uniformizable) or optimal embeddings is still open. It is already difficult to formulate what a "good" embedding is, since the choice of embedding for a statement is also influenced by embeddings of another statements connected in the PRDG, and vice versa. The problem is often viewed as a step in data alignment problem [42, 67, 68], which is slightly different since the target of alignment is arrays instead of iteration space.

5.4 Heuristics for Embedding

We rely on a number of heuristics to find a reasonable embedding. Although a more general solution, which analyze how "good" a choice of embedding, is more desirable, it is left as a future work. The heuristics are similar to those for data alignment, but are different since some of the heuristics, e.g., those that refer to loop indices used in array accesses, are not directly applicable.

The embedding operates on PRDG of programs. We first set the maximum number of dimensions of nodes in the PRDG, as the target dimension. Although it is possible, and sometimes better or even unavoidable, to embed into higher dimensions, we stay with the original number of dimensions. Then we can say those nodes with the highest number of dimensions are already embedded. These nodes form a set $\mathcal{B}$ called base nodes.

Then we seek to find embedding functions for other nodes in the PRDG. To allow transitive construction of such functions, we find a pair $TS(n) = \langle t, s \rangle$, where t is an affine function, and s is a node n in the PRDG, for each node. The function t is the base transform, and is composed with the transform of s: composition target. The final embedding transformation σ is computed by composing t with the embedding transform of s, and hence is recursively defined. This requires that when finding the pairs $\langle t, s \rangle$, the path following s does not form a cycle.

There are some simple cases where a "good" embedding can be found from dependences. The following heuristics are cases where a candidate can be found by inspecting an edge, provided the dependence function is bijective.

- **Dependence on base nodes:** If a node $n \notin \mathcal{B}$ depends on $b \in \mathcal{B}$, then the dependence function itself is a good embedding transformation. For example, if there is a dependence $(i \to i, N)$, then after transforming the node with the same function, the dependence becomes $(i, j \to i, j)$; a uniform dependence.

- **Dependence to a node with higher dimensions:** With a similar argument as the above, dependence functions to a node with higher dimensions is a good candidate.

- **Dependence from a node with higher dimensions:** If a node with higher dimensions m depends on n with function f, then an embedding transformation t on n where $t \circ f$ has the lowest rank is a good candidate. The rank of a dependence is the rank of the linear part of the dependence in matrix representation. For example, if a node depends on n with dependence $(i, j \to i)$, $t = (i \to i, 0)$ is preferred over $(i \to 0, i)$. The former leads to dependence $(i, j \to i, 0)$, which is rank 1, but the latter leads to $(i, j \to 0, i)$; a rank 2 dependence.

We first apply these heuristics, in the order shown above, to find some of the embedding transformations. In all cases, the composition target s is the node with higher number of dimensions in the edge considered. Note that the chain of compositions will never have a cycle, since the direction is always from lower to higher dimensions. If a heuristic give multiple candidates, then one is selected by another set of heuristics discussed later.

Next, a second set of heuristics is used to find safe embedding transforms (i.e., those do not introduce cycles.) We ensure the safety by only introducing embedding transforms where the target is a node that

already have a transform, including those introduced by the following set of heuristics. Since all targets are in higher dimensions after the application of first set of heuristics, introducing additional transforms that do not decrease in dimensions to a node that already has a transform is safe.

- **Dependence on node with embedding:** If n has an edge e to m where $TS(m)$ is already defined, with a bijective function to the space with the same or higher number of dimensions, then the dependence function is a candidate embedding transform.
- **Dependence from node with embedding:** If n has an edge e from m where $TS(m)$ is already defined, with a dependence function that admits an inverse (in context of the dependence domain), then the inverse is a candidate embedding transform.

The above set of heuristics are actually sufficient to find embedding transforms for most nodes. If some node do not have a transform, then the embedding is completed by specifying a legal embedding, which can trivially be found. We use the following to find an embedding.

- If there is another node m with $TS(m)$ already defined, use identity function as the transform, and set m to be the composition target. If the input PRDG has a single weakly connected component, there is at least one such m.

In the application of the heuristics above, some nodes may find multiple edges that match a heuristic. In such cases, we need to select an edge to use.

- If the linear part of the functions are the same, any edge will work.
- If the linear part is not identical, then select an edge that gives the lowest dependence after transformation.

As a secondary optimization, select an edge to some boundary in the higher dimensional space among those that have the same linear part.

The heuristics above, when combined, forms a working algorithm that can find embedding transforms for all nodes. The resulting algorithm is heuristic driven, and the heuristics used are rather simple. However, as we show later in the next section, these set of heuristics are sufficient to find good enough embeddings for `PolyBench` benchmarks.

5.5 "Uniform-ness" of `PolyBench`

In this section, we evaluate the "uniform-ness" of `PolyBench` to by measuring how many of the benchmarks can be uniformized by applying the methods described above. The program is said to be uniform if all dependences in its PRDG, excluding input dependences, are.

Table 5.1 shows if dependences in a benchmark is completely uniform in three stages:

- Uniform at Start: We take the `Alpha` program extracted from C programs, and perform a form of constant propagation to inline constants and scalar variables. Then we construct the PRDG of a program, and check if it is uniform.
- Uniform after Embedding: The PRDG after applying the embedding described in Section 5.3 is analyzed.
- Uniform after Pipelining: The PRDG after applying uniformization by pipelining described in Section 5.1 is analyzed.

We found 5 benchmarks with bugs or questionable implementations that make a computation known to be uniformizable not, or the other way around. Excluding these benchmarks, 21 out of 25 benchmarks are completely uniform after pipelining.

Of the remaining four, three benchmarks can technically be handled. These three benchmarks `correlation`, `covariance`, and `ludcmp` share a common property that some matrix is computed in the first phase of the kernel, and then used in the next phase with transpose-like dependences. Transpose-like dependences are dependences where the linear part is some permutation of the indices. Such dependences are not idempotent and cannot be handled by pipelining. However, it is possible to decompose the program into phases such that the values computed in the former can be used as inputs to the latter. Then the problematic dependences become input dependences, and the program remains uniform. This is obviously not an ideal solution, as it significantly limits the design space.

This leaves `durbin` as the only remaining benchmark, which is known to be difficult to parallelize. In fact, PLuTo will not parallelize this kernel, and it can only tile one of the dimensions.

In conclusion, we found a significant portion of `PolyBench` to be uniformizable, albeit with heuristic heavy embedding, and one of the simplest technique for uniformization. Although no general claims can be made from the study of `PolyBench`, we believe that 84% of `PolyBench` is significant.

5.6 Retaining Tilability after Pipelining

Let us first introduce the notion of *dependence cones*:

- f^* denotes the dependence cone; the smallest polyhedral cone of a dependence f, defined over domain $\mathcal{D}$, containing its range $f(\mathcal{D})$.
- A dependence cone f^* is said to be "pointed" if there exists a vector π such that $\forall x \in f^*, \pi \cdot x > 0$.
- Dependence cone for a set of dependence is the smallest polyhedral cone that contains the union of all ranges.

Table 5.1: Uniform-ness of `PolyBench/C 3.2` [84]. Excluding the four benchmarks with bugs and questionable implementations, 21 out of 25 (84%) can be fully uniformized.

Benchmark	Uniform at Start	Uniform after Embedding	Uniform after Pipelining
`correlation`			
`covariance`			
`2mm`			✔
`3mm`			✔
`atax`			✔
`bicg`	✔	✔	✔
`cholesky`			?[2]
`doitgen`		✔	✔
`gemm`	✔	✔	✔
`gemver`			✔
`gesummv`		✔	✔
`mvt`	✔	✔	✔
`symm`			?[3]
`syr2k`	✔	✔	✔
`syrk`	✔	✔	✔
`trisolv`			✔
`trmm`			?[3]
`durbin`			
`dynprog`			?[4]
`gramschmit`			✔
`lu`			✔
`ludcmp`[1]			
`floyd-warshall`			✔
`reg_detect`		✔	✔
`adi`			?[5]
`fdtd-2d`		✔	✔
`fdtd-apml`		✔	✔
`jacobi-1d-imper`	✔	✔	✔
`jacobi-2d-imper`	✔	✔	✔
`seidel-2d`	✔	✔	✔

[1] `ludcmp` in `PolyBench` is actually LU decomposition followed by forward substitution.

[2] `cholesky` is not uniformized due to a rather strange implementation of Cholesky decomposition. The computation itself is known to be uniformizable.

[3] `symm` and `trmm` do not correctly implement their BLAS equivalent. Correct implementation (and also the incorrect `symm`) is uniform.

[4] `dynprog` is uniformizable, but only due to a bug in its implementation. Correct implementation of Optimal String Parenthesization is not uniformizable.

[5] `adi` is uniformizable, but contains a bug that makes it an incorrect implementation of Alternate Direction Implicit method. With the bug fixed, it is still uniformizable.

Van Dongen and Quinton [117] show that if the dependence cone formed by the set of dependences in the program is pointed, then the extremal rays of this cone can be used as uniformization vectors to ensure that the resulting program also has a pointed dependence cone. This simple result going back more than 25 years is, in hindsight, all we need to ensure tilability after uniformization.

If the dependences before uniformization is such that the program is tilable, then it is tilable after uniformization if the dependences are uniformized with non-negative uniformization vectors. Also following from the same assumption, the dependence cone for all dependences in the starting program is at most the entire non-negative orthant. Thus, it immediately follows that the extremal rays of this cone are non-negative vectors, and thus satisfying the condition.

In the original article, published before the invention of tiling, the dependence cone was used to ensure that the choice of uniformization vectors maintain "schedulability" of system of affine recurrences (which is a subset of `Alpha`) by affine schedules. We have shown that this result carries over to "tilability", and this is rather obvious in hindsight.

5.7 Discussion

In fact, Roychowdhury [104] has shown that all affine dependences can be uniformized, provided any number of dimensions can be added to the iteration space. Although it is not practical to add many dimensions, we may say that no affine dependences are truly affine.

In this chapter, we have shown that even with uniformization by pipelining, most of `PolyBench` can be uniformized. Those that cannot be uniformized by pipelining all share a common pattern of transpose-like dependences, and could be handled with a minor extension to our heuristics.

Assuming uniform dependences, even for a subset of the dependences, can significantly simplify the formalization of analyses and transformations. One may get an impression that only handling uniform dependences is too restrictive, but as we have shown, most affine dependences can easily be uniformized. Although affine dependences are the most general class of dependences that the polyhedral model can handle, we believe that it is important to explore a more restrictive class of dependences that permits more powerful analyses and transformations.

Chapter 6

Memory Allocations and Tiling

In this chapter, we discuss memory allocations for tiled programs, especially for the case when tile sizes are not known at compile time. When the tile sizes are parameterized, most techniques for memory allocations [25, 69, 90, 115] cannot be used due to the non-affine nature of parameterized tiling. However, parametric tiling combined with memory re-allocation is no use if we cannot find a legal allocation for all legal tile sizes.

One approach that can find memory allocations for parametrically tiled programs is the Schedule-Independent Storage Mapping proposed by Strout et al. [112]. For programs with uniform dependences, schedule-independent memory allocation finds memory allocations that are legal for any legal execution of the program, including tiling by any tile size. We present a series of extensions to schedule-independent mapping for finding legal and compact memory allocations for polyhedral programs with uniform dependences.

6.1 Extensions to Schedule-Independent Storage Mapping

The Schedule-Independent Storage Mapping is based on what are called Universal Occupancy Vectors (UOVs) that characterize when a value produced can safely be overwritten. In the remainder of this section, we use UOV-based allocation to refer to schedule-independent storage allocation.

The method makes simplifying assumptions that can lead to inefficient allocations in practical cases. In particular, the assumption that the dependence pattern is same for all points in the iteration domain often leads to inefficiencies due to dependences at the boundaries.

In this section, we extend the UOV-based allocation in the following aspects:

- Handling of imperfectly-nested loops. The original method assumed perfectly nested programs. We extend the method by taking statement orderings, often expressed in the polyhedral model as constant dimensions, into account.

- Refined trivial UOV construction. We show certain properties of dependences and UOVs to reduce the number of dependences to be considered when finding an UOV. This can significantly improve the quality of the trivial UOV (a legal UOV found by a simple method), and also narrow the search space that needs to be explored when the optimal UOV is obtained using the dynamic programming presented by Strout et al. [112].

- We also present a very efficient method, without any search, to find the shortest UOV (in Manhattan distance as we define in Section 6.1.3) when UOV-based allocation is used in a specific context, namely tiled loop programs.

6.1.1 Universal Occupancy Vectors

We first present an overview of Universal Occupancy Vectors (UOVs), and a memory allocation strategy based on UOVs [112]. This approach works for uniform dependence programs, and an important property is that the computed memory allocation is valid for any legal schedule, including tiled execution.

UOV is a vector that characterizes when a value may safely be overwritten. Given the set of dependences $\mathcal{I}$, and an iteration point z, let X be the set of points that depends on z by a dependence in $\mathcal{I}$. Then a vector v is a legal UOV if the point $z' = z + v$ depends either directly or indirectly on all points $x \in X \setminus \{z'\}$ by compositions of the dependences in $\mathcal{I}$. In other words, the point z' must transitively depend on all points that directly use the value produced at z. One exception is the point z' that can also be a consumer of the value produced at z, since most machine models assume that at any given time step, reads happen before writes. Because z' depends on all uses of z, *any* legal schedule would have executed all uses of z, and hence the value may safely be overwritten by z'. Since the dependence pattern is assumed to be the same at all points in the statement domain, the choice of z is not relevant.

UOV is formulated for a set of dependences, and we say UOV with respect to dependence(s) X when the scope of dependences considered are limited to X.

Once the UOV is computed, the memory allocation that corresponds to a projection along the UOV is a legal memory allocation. If the UOV crosses more than one integer points, then an array that corresponds to a single projection is not sufficient. Instead, multiple arrays are used in turn, implemented as modulo factors. The necessary modulo factor is the GCD of elements of the UOV.

The trivial UOV; a valid UOV, which may not be optimal; is computed as follows.

1. Collect the set of dependences for a statement S.
2. Add the *dataflow* vectors of the functions in the set.

The above follows from a simple proposition shown below, and an observation that the dataflow vector of a dependence is a legal UOV with respect to that dependence.

Proposition 6.1 (Sum of UOVs) *The sum of legal UOVs for two sets of dependences is a legal UOV for the combined set of dependences.*

Proof Let the UOVs for two sets of dependences $\mathcal{U}$ and $\mathcal{V}$ respectively be u and v. Then the value produced at z is dead when $z + u$ can legally be executed with respect to the dependences

in $\mathcal{U}$, and similarly for $\mathcal{V}$ at $z+v$. Since there is a path from z to $z+u+v$ by following the edges $z+u$ and $z+v$ (in either order), the value produced at z is guaranteed to be used by all uses, $z+u$ and $z+v$, when $z+u+v$ can legally be executed. ∎

6.1.2 Limitations of UOV-based Allocation

Allocations based on UOVs have a strong property: schedule-independence. Here, the schedule is not limited to affine schedules in the polyhedral model, and time stamps to each operation can be assigned arbitrarily, as long as they respect the dependences.

In the usual flow of compilation, dependences are across multiple statements. Affine scheduling in polyhedral compilation finds affine functions that map each of these statements to a common space, where time stamps of instances of all statements can be compared using lexicographical order. Therefore, the allocation is, strictly speaking, not legal for *any* schedule when put in the context of polyhedral compilation. Instead, UOV-based allocation is only applicable when all statements are taken to this common space.

For example, let two statements A and B both scheduled with affine function $(i, j \to i, j)$ have a dependence $(i, j \to i, j-1)$ from A to B in the scheduled space. If we change the schedule of A to $(i, j \to i, j+1)$, it is executed one time step later than before, and the dependence becomes $(i, j \to i, j-2)$ in the scheduled space. Now, an UOV computed based on the former dependence is clearly invalid. Moreover, changes in affine schedules can turn a uniform dependence to an affine dependence and vice versa.

Thus, when used in the polyhedral compilation flow, UOV-based allocation is schedule-independent if the same schedule is applied to all statements.

6.1.3 Optimal UOV without Iteration Space Knowledge

The optimality of UOVs without any knowledge of size or shape of the iteration space is captured by the length of the UOV. However, the length that should be compared is not the Euclidean length, but the Manhattan distance.

It is easy to show that any multiple, by some integer greater than one, of a legal UOV uses more memory. This is because if some vector $\vec{u}$ is multiplied by an integer $\alpha > 1$, then the number of iteration points that are crossed by the vector $\alpha\vec{u}$ is α times more than that by the vector $\vec{u}$. Since UOV characterizes the distance between two points that can share the same memory location, the number of iteration points that are crossed is directly related to memory usage.

Two UOVs that are not constant multiples of each other are often difficult to compare. For example, memory usage of two allocations based on UOVs $[1, 1]$ and $[2, 0]$ are only parametrically comparable. With $N \times M$ iteration space, the former uses $N + M$ and the latter uses $2N$. The optimal allocation in such case depends on the values of N and M that are not known until run-time.

```
for (t=0:T)
   for (i=0:N)
      A[i] = foo(A[i]);          //S1
   for (i=1:N)
      A[i] = bar(A[i-1], A[i]); //S2
```

Figure 6.1: Imperfectly nested loop

Informally, increasing the Manhattan distance will always increase memory usage by either increasing the GCD, and hence increasing the mod factor, or by increasing the "angle of the projection", and hence increasing the size of the projected space.

6.1.4 UOV in Imperfectly Nested Programs

In the original article, only programs with a single statement are considered. The authors also assume that the loops are perfectly nested. When multiple statements are in a program, the textual order of the statements and/or loops becomes important.

For example, consider the code fragment in Figure 6.1. When array dataflow analysis [30] is applied to the code fragment above, the domains and dependences are (here, we denote dependences as relations, and name label LHS and RHS with the statement name):

- $\mathcal{D}_{S1} = \{t, i | 0 \leq t \leq T \wedge 0 \leq i \leq N\}$
- $\mathcal{D}_{S2} = \{t, i | 0 \leq t \leq T \wedge 1 \leq i \leq N\}$
- $\mathcal{I}_1 = S1[t, i] \rightarrow S2[t-1, i] : i > 0$
- $\mathcal{I}_2 = S1[t, i] \rightarrow S1[t-1, i] : i = 0$
- $\mathcal{I}_3 = S2[t, i] \rightarrow S1[t, i]$
- $\mathcal{I}_4 = S2[t, i] \rightarrow S1[t, i-1] : i = 1$
- $\mathcal{I}_5 = S2[t, i] \rightarrow S2[t, i-1] : i > 1$

In the above example, dependence $\mathcal{I}_3$ is not respected with identity scheduling function, since the producer and the consumer are scheduled at the same time stamp. One possible affine schedule to respect this dependence is:

- $\theta_{S1} = S1[i, j] \rightarrow [i, j, 0]$
- $\theta_{S2} = S2[i, j] \rightarrow [i, j, 1]$

which corresponds to loop fusion of the original program.

Another choice of schedule, which corresponds to the original loop structure is:

- $\theta_{S1} = S1[i,j] \rightarrow [i,0,j]$
- $\theta_{S2} = S2[i,j] \rightarrow [i,1,j]$

These constant dimensions are commonly used in polyhedral scheduling for expressing statement orderings. Notice that the dependences in the schedule space are different between the two choices above. For example, the dependence $\mathcal{I}_3$ becomes the following in the scheduled space:

- $\mathcal{I}_3 = [t,i,1] \rightarrow [t,i,0]$
- $\mathcal{I}_3 = [t,1,i] \rightarrow [t,0,i]$

As discussed earlier, different choice of schedules influence the dependence patterns in the scheduled space. As an obvious consequence, UOV-based allocations are different.

6.1.5 Handling of Statement Ordering Dimensions

When the schedule function has ordering dimensions, the corresponding elements in the UOV require special handling. In the original formulation there is only one statement, and thus every iteration point writes to the same array. When multiple statements exist in a program, the iteration space of a statement is a subset of the combined space, and is made disjoint by statement ordering dimensions. Thus, not all points in the common space correspond to a write, and this affects how the UOV is interpreted. It is possible for two statements to have overlapping spaces when the dependences are such that the statement order is irrelevant. However, we assume that the spaces are disjoint to simplify our presentation.

When all points in the iteration space write to the same array, the elements in the UOV directly correspond to the number of values that must be live. For example, consider the following program:

```
for (i=1:N)
   A[i] = A[i-2]; //S1
```

A trivial UOV of $S1$ is the vector [2], and at least 2 scalars are required to preserve a value computed i until its last use; $i+2$, because there is another value computed and to be stored at $i+1$.

However, the situation is different for statement ordering dimensions. Consider the following:

```
A = S1()  //S1
B = S2(A) //S2
C = S3(A) //S3
```

Note that this example does not even have a loop, but the iteration space is one-dimensional, due to constant ordering dimensions. The dependences are $S2[1] \rightarrow S1[0]$ and $S3[2] \rightarrow S1[0]$, and the constant parts are $[-1]$ and $[-2]$. Although the shortest valid UOV is [2], it is obvious that a single scalar is sufficient for the above program. This is because the iteration points that depend on $S1$ are not instances of $S1$, and hence they are allocated separately.

Because we assume all statement domains are made disjoint by statement ordering dimensions, a vector of any length along the constant dimensions will only cross at most one iteration point of a statement. We can safely say that the elements of a UOV that correspond to statement ordering dimensions may only be 1, 0, or -1. In the corresponding dimensions of the dependence vector, any positive value is replaced by 1, and negative value is replaced by -1, when constructing the UOV.

In addition, the ordering dimensions may be ignored in the final memory mapping function, computed as the affine/linear function whose kernel is the UOV, since their size is always 1.

6.1.6 Dependence Subsumption for UOV Construction

The trivial UOV, which also serves as the starting point for finding the optimal UOV, is found by taking the sum of all dependences. However, this formulation may lead to significantly inefficient starting point. For example, if two dependences $[i] \rightarrow [i-1]$ and $[i] \rightarrow [i-2]$ exist, the former dependence may be ignored during UOV construction since a legal allocation for the latter is already also legal for the former.

We may refine both the construction of the trivial UOV and the optimality algorithm by reducing the set of dependences considered during UOV construction. The optimality algorithm presented by Strout et al. [112] searches a space bounded by the length of trivial UOV using dynamic programming. Therefore, reducing the number of dependences to consider will improve both the trivial UOV and the dynamic programming algorithm.

The main intuition is that if a dependence can be transitively expressed by another set of dependences, then the longest dependence (i.e., the dependence expressed transitively), is the only dependence that needs to be considered. This is formalized in the following lemma.

Lemma 6.2 (Dependence Subsumption) *If a dependence f can be expressed as a composition of dependences in a set $\mathcal{G}$, where all dependences in $\mathcal{G}$ are used at least once in the composition, then a legal UOV with respect to f is also a legal UOV with respect to the dependences in set $\mathcal{G}$.*

> **Proof** Given a legal UOV with respect to f, a value produced at z is preserved at least until z' defined by $f(z') = z$, can be executed. Let the set of dependences in $\mathcal{G}$ be denoted as g^x, $1 \leq x \leq n$, where n is cardinality of $\mathcal{G}$. Since composition of uniform functions is associative and commutative, there is always a function g^* obtained by composing dependences in $\mathcal{G}$, such that $f = g^* \circ g^x$ for each x. Thus, all points z'', $g^x(z'') = z$, are executed before z' for all x. Therefore, a legal UOV with respect to f is guaranteed to preserve the value produced at z until all points that directly depend on z by a dependence in set $\mathcal{G}$ have been executed. ∎

Finding a composition in the above can be implemented as an integer linear programming problem. The problem may also be viewed as determining if a set of vectors are linearly dependent when restricted to

positive combinations. The union of all sets $\mathcal{G}$, called subsumed dependences, found in the initial set of dependences can be ignored when constructing the UOV.

Applying Lemma 6.2 may significantly reduce the number of dependences to be considered. However, the trivial UOV of the remaining dependences may still not be the shortest UOV. For example, consider dependence vectors $[-1,-1],[-1,1],[-1,0]$. Although the vectors are independent by positive combinations, the trivial UOV $[3,0]$ is clearly longer than another UOV $[2,0]$. Further reducing the set of dependences to consider requires a variation of Lemma 6.2 that allows f to be also a composition of dependences. This leads to complex operations, and the dynamic programming algorithm for finding optimal UOV in the original article [112] may be a better alternative. Instead of finding the shortest UOV in the general case, we show that such UOV can be found very efficiently for a specific context, namely tiling.

6.2 UOV-based Allocation for Tiling

UOV-based allocation give legal mappings even for schedules that cannot be implemented as loops. For example, a run-time scheduler that selects an iteration to execute randomly, among those iterations where all required values are computed, can use UOV-based allocation. However, this is obviously an overkill if we only consider schedules that can be implemented as loops.

When UOV-based allocation is used in the specific context of tiling, the initial UOV may be further optimized. If we know that the program is to be tiled, we can add *dummy* dependences to restrict the universality of the storage mapping, while maintaining tilability. In addition, we may assume that the dependences are all non-positive (for the tilable dimensions) as a result of pre-scheduling step to ensure the legality of tiling. For the remainder of this chapter, the "universe" of UOVs are one of the following restricted universes: fully tilable, fully sequential, and mixed sequential and tilable.

Theorem 6.3 (Shortest UOV in Fully Tilable Space) *Given a set of dependences $\mathcal{I}$ in a fully tilable space, the shortest UOV u for tiled execution is the element-wise maxima of dataflow vectors of all dependences in $\mathcal{I}$.*

> **Proof** Let the element-wise maxima of all dataflow vectors be the vector m, and f^m be a dependence with dataflow vector m. We introduce the following *dummy* dependences $f^d; f^d(z) = z - u^d$ where u^d is the d-th unit vector. These dependences do not violate the fully-permutable property of the tilable space, and therefore the resulting space is still tilable. For all dependences in $\mathcal{I}$ there exists at least a sequence of compositions with the dummy dependences to transitively express f^m. Using Lemma 6.2, the only dependence to be considered in UOV construction can be reduced to f^m, which has the vector m as its trivial UOV.

It remains to show that no UOV shorter than m exists for the set of dependences $\mathcal{I}$. The shortest UOV is defined by the closest point from z, where all uses of an iteration z can reach by following the dependences. Since the choice of z does not matter, let us use the origin, $\vec{0}$, to simplify our presentation. This allows us to use the dataflow vectors interchangeably with coordinate vectors.

Then, the smallest hyper-rectangle that contains m includes all $\mathcal{I}$, and all bounds of the hyper-rectangle are touched by at least one dependence. Since all dependences in a fully tilable space are restricted to have non-negative dataflow vectors, no points within the hyper-rectangle can be reached by following dependences. Thus, it is clear that m is the closest common point that can be reached by those that touch the bounds. ∎

The basic idea of inserting dummy dependences to restrict the possible schedule can be used beyond tilable schedules. One important corollary for sequential execution is the following.

Corollary 6.4 (Shortest UOV for Sequential Execution) *Given a set of dependences $\mathcal{I}$ in an n-dimensional space where lexicographic scan of the space is a legal schedule, the shortest UOV u for lexicographic execution is the vector $[m_1 + 1, 0, \cdots, 0]$ where m is the lexicographic maxima of the dataflow vectors of all dependences in $\mathcal{I}$.*

Proof For sequential execution, we may introduce dummy dependences of the form $[1, *, \cdots, *]$ and/or $[0, .., 0, 1, *, \cdots, *]$, where $*$ can take any integer value. Since the leading dimensions before any $*$ is either 0 or 1, the resulting dependence is always to a lexicographically preceding iteration, and thus the dependences of this form are always respected with lexicographic execution. Note that the dummy dependences introduced in Theorem 6.3 are included in the above.

Using the dummy dependences, any dependence the first element of whose dataflow vector is smaller than the maxima is subsumed according to Lemma 6.2. For the remaining dependences, there are two possibilities:

- We may use dummy dependences of the form $[1, *, \cdots, *]$ to let a dependence with dataflow vector $[m_1 + 1, 0, \cdots, 0]$ subsume all remaining dependences.
- We may use m as the UOV following Theorem 6.3.

It is obvious that when the former options is used, $[m_1+1, 0, \cdots, 0]$ is the shortest. The optimality of the latter case follows from Theorem 6.3. Thus, the shortest UOV is the shortest among these two options. When the lengths of two possibilities are compared in Manhattan distance, the former possibility is at least as short as the latter, and hence, we may always use the former. ∎

The following corollary can trivially be established by the combination of the above.

Corollary 6.5 (Shortest UOV for Sequence of Tilable Spaces) *Given a set of dependences $\mathcal{I}$ in a space where a subset of the dimensions are tilable, and lexicographic scan is legal for other dimensions, the shortest UOV u for tiled execution of the tilable space, and sequential execution of the rest is the combination of vectors computed for each contiguous subset of either sequential or tilable spaces.*

Note that the above corollary only takes effect when there are sequential subsets with at least two contiguous dimensions. When a single sequential dimension is surrounded by tilable dimensions, its element-wise maxima and lexicographic maxima are equivalent.

Using the above, the shortest UOV for sequential, tiled, or a hybrid combination, can be computed very efficiently.

6.3 UOV-based Allocation per Tile

In the above discussion, the allocations were for tilable iteration spaces. However, UOV is purely a property of the dependences, and its correctness does not rely on domains of the iteration space. Therefore, allocations computed using UOV can be directly used for computing allocations per tile, or any set of tiles.

In the context of shared memory architecture, allocating memory per tile does not make sense. However, for distributed memory architecture, where memory must be partitioned among nodes, allocation for some subset of the iteration space becomes necessary. UOV-based allocation can be used to partition the memory by allocating memory for the set of tiles to be executed by a processor.

Furthermore, the allocation for set of tiles may be further optimized with UOV analysis by treating each tile as an iteration, and dependences between tiles as dependences. The resulting UOV across tiles indicate when the allocation for a tile may safely be reused by another tile.

6.4 UOV Guided Index Set Splitting

In polyhedral representation of programs, or even in `Alpha` programs written equationally, there are always boundary cases that behave differently from the rest. For instance, the first iteration of the loop may read from inputs, where successive iterations use values computed by previous iterations.

In the polyhedral model, memory allocation is usually computed for each statement. With pseudo-projective allocations, the same allocation must be used for all points in the statement domain. Thus, dependences that only exist at the boundaries influence the entire allocation.

For example, consider the code in Figure 6.1. The value produced by `S1[t,i]` is last used by `S2[t,i+1]` for $i > 0$. However, the value produced by `S1[t,0]` is last used by `S1[t+1,0]`. Thus, the memory allocation

for `S1` must ensure that a value produced at `[t,i]` is live until `[t+1,i]` for all instances of `S1`. This clearly leads to wasteful allocations, and our goal is to avoid them.

One solution to the problem is to apply a form of *index set splitting* [38] such that the boundary cases and common cases have different mappings. In the example above, we wish to use piece-wise mappings for `S1[t,i]` where the mapping is different for two disjoint sets $i = 0$ and $i > 0$. This reduces the memory usage from an array of size $N + 1$ to 2 scalars.

Once, the *pieces* are computed, application of piece-wise mappings can done through splitting the statements (nodes) as defined by the pieces, and then applying a separate mapping to each of the statements after split. Thus, the only interesting problem that remain is finding meaningful splits.

In this section, we present an algorithm for finding the split with the goal of minimizing memory usage. The algorithm is guided by Universal Occupancy Vectors and works best with UOV-based allocations. The goal of our index set splitting is to isolate boundaries that require longer lifetime than the main body. Thus, we are interested in a sub-domain of a statement with a different dependence pattern compared to the rest of the statement. We focus on boundary domains that contain at least one equality. The approach may be generalized to boundary planes of constant thickness using Thick Face Lattices [40].

The original index set splitting [38] aimed at finding better schedules after the split. The quality of the split is measured by its influence on possible schedules: if having different scheduling functions to each piece helps in finding a better schedule.

In our case, the goal is different. Our starting point is the program after affine scheduling, and we are now interested in finding memory allocations. When the dependence pattern is the same at all points in the domain, splitting cannot improve the quality of the memory allocation. Since the dependence pattern is the same, the same memory allocation will be used for all pieces (with a possible exception of the cases when the split introduces equalities or other properties related to shape of the domains). Because two points that may have been in the nullspace of the projection may now be split into different pieces, the number of points that can share the same memory location may be reduced as the result of splitting.

Thus, as a general measure of quality, we seek to ensure that a split influences the choice of memory allocation for each piece. The obvious case when splitting is useless is when a dependence function at a boundary is also in the main part. We present Algorithm 6.1 based on this intuition to reduce the number of splits.

The intuition of the algorithm is that we start with all dependences with equalities in their domain as candidate pieces. Then we remove some of the dependences where splitting does not improve the allocation from candidate pieces. The obvious case is when the same dependence function exists in the non-boundary cases (i.e., dependences with no equalities in their domain). In addition, more sophisitcated exclusion is performed using Theorem 6.3.

Although the algorithm is tailored towards UOV-based allocation by using UOV as an abstraction to find useless splits, it may still be used for non-uniform programs. It may also be specialized/generalized by adding more rules to eliminate dependences to Step 2. This requires a method similar to Lemma 6.2 for other memory allocation methods.

6.5 Memory Usage of Uniformization

In uniformization by pipelining discussed in Chapter 5, an affine dependence was replaced by sequences of uniform dependences. Since the only use of the uniformization variable—the variable introduced by uniformization—is the propagation dependence, the uniformization vector becomes the UOV for the variable.

Although the identity dependence to uniformization variable that replaced the affine dependence is another use of this variable, it can be ignored here. This identity dependence can always be handled by the statement ordering that ensures the propagation happens before the use. Thus, the UOV used for the allocation is exactly the uniformization vector, and hence only one element is allocated for propagation of a value. Therefore, no extra memory is required for propagation.

Additionally, since both producer and consumer iterations of the propagation are mapped to exactly the same memory location, there is no need to actually perform the copy. Thus, uniformization, which may initially appear inefficient, can be implemented with almost no overhead.

6.6 Discussion

In this chapter, we have presented a series of techniques for finding allocations that work for parametric tiling. Building on the result of Chapter 5 that showed many affine control programs are uniformizable, we have extended Schedule-Independent Storage Mapping [112] to handle various cases.

By combining all the proposed extensions, we can find compact allocations that can be used for any (legal) tile sizes. In addition, tailoring UOV-based allocation to a specific context of tiling allows us to find the shortest UOV without any search. Although finding the optimal allocation given constant problem sizes would still require the dynamic programming algorithm, we focus on cases where problem sizes are run-time parameters. In such cases, our allocations are as compact as UOV-based allocation can get.

Algorithm 6.1 UOV-Guided Split

Input:

$\mathcal{I}$: Set of dependences that depend on a statement S. A dependence is a pair $\langle f, D\rangle$ where f is an affine function and D is a domain.

Output:

$\mathcal{P}$: A partition of D_S, the domain of S, where each element defines a piece of the split.

Algorithm:
We first inspect the domain of dependences in $\mathcal{I}$ to detect equalities.
Let

$\mathcal{I}_b$ be the set of dependences with equalities, and

$\mathcal{I}_m$ be the set of those without equalities.

Then,

1. **foreach** $\langle f, D\rangle \in \mathcal{I}_b$,

   if $\exists \langle g, E\rangle \in \mathcal{I}_m; f = g$ **then** remove $\langle f, D\rangle$ from $\mathcal{I}_b$

2. Further remove dependences from $\mathcal{I}_b$ using the following *if applicable*:

   (a) Theorem 6.3 and its corollaries. The following steps are only for Theorem 6.3.

   Let m be the element-wise maxima of dataflow vectors in $\mathcal{I}_m$.
   foreach $\langle f, D\rangle \in \mathcal{I}_b$
   - Let v be the dataflow vector of f.
   - **if** $\forall_i : v_i \leq m_i$ **then** remove $\langle f, D\rangle$ from $\mathcal{I}_b$.

3. Group the remaining dependences in $\mathcal{I}_b$ into groups $\mathcal{G}_i$, $0 \leq i \leq n$, where $\forall X, Y \in \mathcal{G}_i; X_f(\mathcal{D}_X) \cap Y_f(\mathcal{D}_Y) \neq \emptyset$. In other words, group the dependences with overlapping domains in the producer space.

4. **foreach** $i \in 0 \leq i \leq n$, $\mathcal{P}_i = \bigcup_{\forall X \in \mathcal{G}_i} X_f(\mathcal{D}_X)$

5. **if** $n \geq 0$ **then** $\mathcal{P}_{n+1} = \mathcal{D}_S \setminus \bigcup_{i=0}^{n} \mathcal{P}_i$ **else** $\mathcal{P}_0 = \mathcal{D}_S$

Chapter 7

MPI Code Generation

In this chapter, we present our approach for generating distributed memory parallel code from polyhedral representations. We target distributed memory parallelization with Message Passing Interface (MPI) and C.

In the last decade, a number of techniques were developed to efficiently parallelize polyhedral representations for multi-core processors. The state-of-the-art techniques use tiling as a method to expose parallelism, and to control locality, for efficient execution [16].

The performance of tiled code is hugely influenced by the choice of tile sizes [22, 60, 93, 99, 100, 102, 114, 132]. This led to the development of series of techniques for parameterized tiled code generation [43, 44, 55, 54, 53, 101]. We may now say that automatic parallelization of polyhedral programs for shared memory is a largely solved research problem although many engineering problems still remain.

However, shared memory architectures cannot be expected to scale to large number of cores. One of the reasons being cache coherency; as a number of cores that share a memory increases, maintaining cache coherency becomes increasingly expensive. It is then a natural flow to target distributed memory code generation.

Distributed memory parallelization introduces a number of new problems. In addition to the partitioning of computation, we must address two additional problems: data partitioning and communication. These two problems are not visible for shared memory parallelization, since the shared memory conveniently hides these issues from the software layer. Moreover, the problem of tile size selection is even more critical in distributed memory parallelization. In addition to locality and load balancing concerns which are also faced by shared memory parallelization, the volume and frequency of communication is also influenced by tile sizes.

In this chapter, we present a code generation technique for efficiently parallelizing polyhedral applications using MPI. How we address the problems described above is summarized below:

- Computations are partitioned using wave-front of tiles. The partitioning of computation is identical to that used for shared memory by PLuTo [16]. However, unlike PLuTo or other related work [8, 15, 21, 59], we allow parametric tiling.

- Data Partitioning is carried by extensions to UOV-based allocation presented in Chapter 6. This allows memory to be allocated for sets of tiles, and also ensures legal memory allocation for parametric tile sizes. It is important to note that we do not require UOV-based allocation in our code generator, which will work with any legal memory allocation for parametric tiling.

- Communications are greatly simplified by our assumption that the dependences that cross processor boundaries are uniform. This builds on our analyses of `PolyBench` in Chapter 5 that shows most benchmarks can be completely uniformized.

  With uniform dependences, many questions related to communication; how to find processors that need to communicate, how to pack/unpack buffers, how to overlap communication with computation; can easily be answered.

Our approach builds on D-Tiling [53]; a parameterized tiled code generation technique for shared memory; and supports parametric tile sizes as well. In the following, we first describe key elements of D-Tiling that we need to described our extensions, and then present how we address the three problems in detail.

7.1 D-Tiling: Parametric Tiling for Shared Memory

Our approach extends upon the state-of-the-art techniques for parameterized tiled code generation [44, 53]. The two techniques are nearly identical, but our presentation is based on the approach by Kim [53] called D-Tiling. Let us first describe the key ideas in D-Tiling in order to explain our extensions.

Since parametrically tiled programs do not fit the polyhedral model, current techniques *syntactically* modify the loop nests after code generation. The code generation works differently for sequential and parallel execution of the tiles. This is because parallel execution of tiles require *skewing* of the tiles, which can be represented as affine transformations if the tile sizes are fixed, that are no longer polyhedral transformations.

For sequential execution of the tiles, D-Tiling takes a d-dimensional loop nest as an input, and uses two different algorithms to produce what are called *tile loops* and *point loops* that are d-dimensional each. Tile loops are loop nests that visit all the *tile origins*, the lexicographical minima of a tile. Points loops are loop nests, parameterized by tile origins, that visit all points in a tile. The final output is simply the two loops composed; point loops as the body of tile loops.

For parallel execution, point loops remains the same, but the tile loops are generated using a different algorithm. After applying tiling, the wave-front of tiles that are executed in parallel are usually those on a hyper-plane characterized by normal vector $\vec{\mathbf{1}} = [1, 1, ..., 1]$.

This is necessary for parametric tiles, because parametric tiling falls outside the polyhedral model. The first iterator *time* corresponds to the global time step of wave-front execution, controlling which wave-front hyper-plane should be executed. Usually, wave-front hyper-planes characterized by normal vector $\vec{\mathbf{1}} = [1, 1, ..., 1]$ is used. This can be viewed as the hyper-plane with 45 degrees slope in all dimensions. This hyper-plane is always legal to be executed in parallel, and can be further optimized by replacing 1 with 0 for dimensions where no dependences are carried across tiles. In the following, we refer to the *time* as the wave-front time.

```
//computation of wave-front time step start and end
start = ...
end = ...
//time loop that enumerates wave-front hyper-planes
for (time = start; time <= end; time++)
   //loops to visit tile origins for wave-front time step "time"
   forall (ti1 = ti1LB; ti1 <= ti1UB; ti1+=ts1) {
      ...
         // last tile origin is computed as a function of
         // current wave-front time step and other tile origins
         tid = getLastTI(time, ti1, ti2, ...)
         //guard against outset
         if (tidLB <= tid && tid <= tidUB) {
             //point loops
             ...
         }
   }
   #synchronization
```

Figure 7.1: Structure of loop nests tiled by D-Tiling [53] for wave-front execution of the tiles. Bounds on `time`; `start` and `end` are computed using Equation 7.1. The last tile origin, ti_d is computed using ti_1 through ti_{d-1}, and ti_d is not a loop iterator. The function `getLastTI` computes ti_d using Equation 7.2. There is a guard to check if the computed ti_d is not an empty tile, and then the point loop. All loops up to ti_{d-1} can be executed in parallel.

Given such a vector v, the wave-front time step for each tile can be computed as:

$$time = \sum_{k=1}^{d} v_k \left\lfloor \frac{ti_k}{ts_k} \right\rfloor$$

where ti is the vector of tile origins, and ts is the vector of tile sizes.

To simply our presentation, we assume that $v = \vec{\mathbf{1}}$, and use the following:

$$time = \sum_{k=1}^{d} \left\lfloor \frac{ti_k}{ts_k} \right\rfloor \tag{7.1}$$

The above formula will also give the lower and upper bounds of time, *start* and *end*, by substituting ti_k with its corresponding lower bound and upper bound respectively.

Similarly, if the *time* and ti_1 through ti_{d-1} is given, ti_d can be computed as:

$$ti_d = (time - \sum_{k=1}^{d-1} \frac{ti_k}{ts_k}) ts_d \tag{7.2}$$

Using Equations 7.1 and 7.2, D-Tiling produces tile loops that visit tile origins of all tiles to be executed in parallel, parameterized by *time*. The structure of the tile loops produced for tiled execution is illustrated in Figure 7.1.

7.2 Computation Partitioning

The basic source of parallelism is the wave-front of the tiles. After tiling, the wave-fronts of the tiles are always legal to run in parallel. This approach has been shown effective in shared memory, by combining this parallelism with data locality improvements from tiling [16]. The polyhedral representation of loop programs is first transformed according to the PLuTo schedule (we use a variation implemented in Integer Set Library by Verdoolaege [121]). Then we tile the outermost tilable band, and parallelize the wave-front of the tiles.

Since parallelization with MPI is SPMD, the code to implement the partitioning is slightly more complicated than annotating with OpenMP pragmas in the shared memory case. We implement a cyclic distribution of processors among the outermost parallel loop using the processor ID, `pid`, and the number of processors, `numP`, as follows:

```
for (ti1=ti1LB+pid*ts1; ti1<=ti1UB; ti+=ts1*numP)
```

Additionally, it is easy to extend the distribution to *block* cyclic by adding another loop:

```
for (bStart=ti1LB+pid*ts1*blockSize; ti1<=ti1UB; ti+=ts1*blockSize*numP)
   for (ti1=bStart; ti1<=bStart+blockSize*ts1; ti1+=ts1)
```

The transformations above are purely syntactic, and it can be observed that the iterations being visited are unchanged. Since it is a parallel loop, the order does not matter, and hence they are legal transformations.

Block cyclic distribution can become useful, especially in distributed memory platforms, because it can be used to control the ratio of computation per communication, without affecting the locality behavior by the tiles. Although OpenMP supports more dynamic allocations through the `schedule` clause of the `for` work sharing pragma, such as `dynamic` or `guided`, these distributions cannot be implemented statically as loops.

7.3 Data Partitioning

We use the memory allocations presented in Chapter 6 to allocate memory. However, we may use any other memory allocation that is also legal for wave-front execution of parametric tiles. We do not use any properties specific to UOV-based allocations.

In our approach memory is allocated per "slices of tiles". A slice of a tile is the set of all tiles that share a common ti_1, the first tile origin. Since this is the unit of distribution of work among processors, separate memory is allocated for each slice. With block cyclic distribution, a slice is extended to be $\texttt{blockSize} \times \texttt{ts1}$. Since each slice in a block is contiguous, the iteration space of the full block is equivalent to a single slice where the outermost tile $\texttt{ts1}' = \texttt{ts1} \times \texttt{blockSize}$. For simplifying the control structure of the generated code, we allocate memory in each processor assuming every tile is a full tile. Each processor allocates memory for the above slice, multiplied by the number of blocks assigned.

We also extend the slice by the longest dependence crossing processor boundaries to accommodate for values communicated from neighboring tiles. This corresponds to what are often called "halo" regions or ghost cells. The values received from other processors are first unpacked from the buffers to arrays allocated for tiles. Then the computation does not have to be special cased for tile boundaries, since even at the bounds, same array accesses can be used.

7.4 Communication

When we can assume all dependences that cross processor boundaries are uniform, communication is greatly simplified. We first assume that the length of uniform dependences in each dimension are strictly less than the tile size of each dimension. Then we are sure that the dependences are always to a neighboring tile, including diagonal neighbors.

7.4.1 Communicated Values

The values to be communicated are always values produced at the tile facets. The values produced at the tile facet touching another tile must be communicated if the two tiles are mapped to different processors. The facets that need to be communicated may be of a constant thickness if the length of dependence is greater than 1 in the dimension that crosses processor boundaries. We call this thickness the communication depth, as it corresponds to how deep into the neighboring tiles the dependences reach.

The depth is defined by the maximum length of a dependence in the first tiled dimension. For example, dependences $[-2, 0]$ and $[-2, -2]$ are both considered to have length 2 in the first dimension. The memory allocated are extended exactly by this depth, and the received values are stored in the "halo" region.

If the memory allocation is legal, then it follows that the values stored in the "halo" regions stay live until their use, even though some of the values may not be immediately used by the receiving tile. Moreover, with the possible exception at the iteration space boundaries, the values communicated are also guaranteed to be used in the next wave-front time step. This follows from the assumption that no dependences are longer than the tile size in each dimension. Therefore, we conclude that the volume of communication is optimal.

The participants of a communication can be identified by the following functions that take the vector of tile origins ti as an input:

$$\texttt{sender}(ti) = [ti_1 - ts_1, ti_2, \cdots, ti_d] \tag{7.3}$$

$$\texttt{receiver}(ti) = [ti_1 + ts_1, ti_2, \cdots, ti_d] \tag{7.4}$$

The above is the solution for one of the most complicated problems related to communication; finding the communicating partners. For programs with affine dependences, much more sophisticated analysis is required [15, 21, 57].

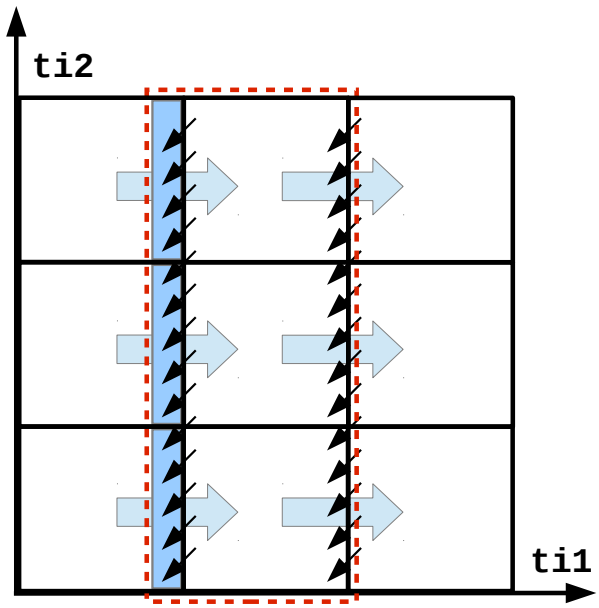


Figure 7.2: Illustration of communications for 2D iteration space with $[-1, -1]$ dependence. The big arrows show the communication, from sender to receiver. The shadowed column (blue) is the tile face being communicated. With greater depth, more columns will be included in the communication in 2D case. The dashed rectangle (red) denotes a slice for which memory is allocated. Note that the upper right corner of a tile, the top-most value of each face, is sent to the horizontal neighbor, but the dependence is from the diagonal neighbor. UOV-based allocation guarantees that this value is never overwritten until its use, and therefore it is safe to send early.

The packing and unpacking of buffers for communicating these variables are implemented as loops after the execution of a tile. The bounds of these loops, and the size of the buffer can be trivially computed, as they are always tile facets of some constant thickness. As an optimization, we allocate a single buffer for multiple statements, if the data types match.

Figure 7.2 illustrates communication for 2D iteration space.

7.4.2 Need for Asynchronous Communication

Conceptually, we may place MPI send and receive calls at the end and the beginning of each tile. With the exception of those tiles at the initial boundary, a tile receives values from neighboring processors before the start of the computation. Similarly, after a tile is computed, values are sent to its neighbor, with the exception of those tiles at the terminating end.

MPI specifies that calls to `MPI_Send` blocks until the buffer for communication has been secured. In most cases, data is copied to system buffer and the control almost immediately returns. However, it should be assumed, for the sake of correctness, that `MPI_Send` is blocking until the corresponding `MPI_Recv` is called.

From the time and receiver functions (Equation 7.1 and 7.4), and also from Figure 7.2, it is clear that the receiver tile is one wave-front time step later than the sender. This means that with naïve placement

of MPI calls, the corresponding receive is not reached until all sends in a wave-front time complete. This clearly can cause a dead lock, when one of the calls to `MPI_Send` actually behaves as a blocking operation.

Therefore, in order to ensure correctness, we must manage buffers ourselves and use asynchronous versions of MPI functions. Even if no dead locks occur, notice that the number of communications "in flight" will increase as the number of tiles increase. The amount of buffer required for the naïve placement of communications corresponds to the maximum number of tiles in a wave-front time step. This is obviously wasteful, and should be avoided.

In addition, we may benefit from overlapping of communication with computation, sometimes called double buffering, by using asynchronous calls. Since data transfers in distributed memory environment typically have significant latency, it is a well-known optimization to hide the latency by computing another set of values as a data transfer is in progress (e.g., [27, 24, 113].) Tiling based parallelization can achieve this overlap if the number of parallel tiles is more than the number of physical processors. After a processor computes a tile, there are more tiles to be computed within the same wave-front time step, allowing overlap.

7.4.3 Placement of Communication

We now address the problem of where the asynchronous communications should be placed to ensure correctness, to reduce the number of in flight communications, and to overlap communication with computation. Let us ignore the problem of code generation for the moment, and assume that communication code can be inserted at the beginning or the end of any tile. We will deal with the problem of code generation separately in Section 7.4.4.

Let us consider a send by a tile identified by its tile origin s, and its receiver, $r = s + [ts_1, 0, \cdots, 0] =$ `receiver`(s), where `receiver` is the receiver in the naïve placement as defined in Equation 7.4. Instead of this naïve placement, we propose to receive the data at $c = r - [0, \cdots, 0, ts_d]$, or in relation to the sender, $c = s + [ts_1, 0, \cdots, 0, -ts_d]$. The new communication pattern is illustrated in Figure 7.3. Note that even though the tile that receives the values is now different, the memory locations where the received buffer is unpacked remain the same.

The tile c is the tile with the same tile origins up to $d-1$ as r, but one wave-front time earlier. Let $v[x : y]$ denote a sub-vector of v consisting of elements x through y. Since c and r only differ in the d-th dimension, $c[1 : d-1] = r[1 : d-1]$. Let the common part be denoted as a vector x, and r be a tile executed at time t, then we observe the following using Equation 7.2, $r_d =$ `getLastTI`(t, x) and $c_d =$ `getLastTI`$(t-1, x)$. Since r was one wave-front time later than s, it follows that c is in the same wave-front time as s.

Furthermore, due to the loop structure of D-Tiling, and how computation is partitioned, we can say that if s is the n-th tile visited by a virtual processor p, then c is the n-th tile visited by the neighboring virtual processor $p+1$, provided that the receive happens also in empty-tiles at the boundaries. Recall that c and

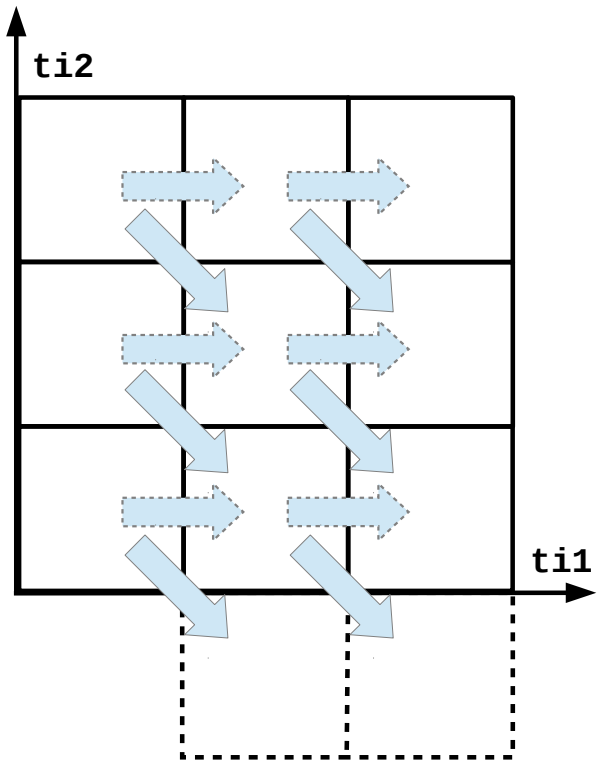


Figure 7.3: Illustration of when the values should be received using the same program structures as in Figure 7.2. The horizontal arrows show the naïve placement, and the diagonal arrows show the optimized placement.

s only differ in the first and the last tile origins, and that the difference $s - c$ is $[ts_1, 0, \cdots, 0, -ts_d]$. The tile loops to iterate ti_2 through ti_{d-1} are common across all processors, and thus the same values are visited by all processors in the same order. Also recall that ti_d is uniquely computed by Equation 7.2. The values of ti_1 for p and $p+1$ differs by $ts1$, and if ts_2 through $ts_d - 1$ are equal, the tile visited by the two processors always differ by $[ts_1, 0, \cdots, 0 - ts_d]$.

Thus, if the communication by s is to be received by c, even a single buffer is sufficient for correct execution. However, this leads to a globally sequentialized execution, unless a processor is only responsible for a single slice at each wave-front. This is because only the first (boundary) processor without any data to receive can proceed to send before receive, and all other processors must wait to receive data before sending. This wait on receive before send propagates to all processors, completely sequentializing the execution.

By increasing the number of buffers to 2 per processor, one for send and one fore receive, each processor can perform both send and receive simultaneously, allowing for parallel execution of tiles. In addition, when the program requires multiple datatypes to be communicated the number of required buffers are multiplied.

However, with 3D or higher iteration spaces, where a slice of tiles contains multiple tiles to be executed in a wave-front time step, 2 buffers are not sufficient for the last physical processor. This is because the send by the last physical processor is not received until the first processor moves to its next slice. Thus, only for the last processor, the size of the send buffer must be multiplied by the number of tiles executed by a slice in one wave-front time step.

Finally, we must also change the placement of receive within a tile to overlap communication with computation. Let us define two functions `prev` and `next` that respectively give the tile executed in previous and next wave-front time step for a given tile origin. These equations can be easily derived from Equation 7.1.

$$\texttt{prev}(ti) = ti - [0, \cdots, 0, ts_d] \tag{7.5}$$

$$\texttt{next}(ti) = ti + [0, \cdots, 0, ts_d] \tag{7.6}$$

The order of operations in a tile ti is as follows:

1. Issue asynchronous receive (`MPI_Irecv`) from `sender`(`next`(ti)).
2. Perform computations for the current tile ti.
3. Wait (`MPI_Wait`) for the send issued at the previous tile to complete.
4. Asynchronously send (`MPI_Isend`) outputs of ti to `prev`(`receiver`(ti)).
5. Wait (`MPI_Wait`) for the issued receive to complete, and unpack buffer.

The key is that instead of performing the receive at the beginning of the tile, as in the naïve approach, receive and unpacking takes place after the computation. Also note that the send always has one tile worth

of computation in between its issue and its corresponding wait. Thus, the communication is overlapped with a tile of computation.

7.4.3.1 Legality

The legality of the above placement can be easily established. The order in which tiles in a wave-front hyper-plane is executed should not matter since they are parallel. Therefore, although some of the parallel tiles may be executed sequentially in the final code due to resource limitations, we may think of virtual processors running at most one tile at each wave-front time step. Then the beginning of the $n+1$-th tile, and the end of n-th tile are equivalent in the final order within a virtual processor. The former case is that of the naïve placement, and the latter is our proposed placement.

7.4.3.2 Beyond Double Buffering

Increasing the number of buffers beyond two will only allow the receive (`MPI_Irecv`) to start early by making buffers available earlier. This does not necessarily translate to performance, if maximum communication and computation overlap is achieved with double buffering. Denoting the time to compute a tile as α, time to transfer outputs of a tile as β, we claim that double buffering sufficient if $\alpha \geq \beta$. When $\beta > \alpha$, one alternative to increasing the number of buffers is to increase α, i.e., increase the tile volume relative to its surface area.

For tiled programs, α and β are both influenced by tile sizes. Since a facet of tiles are communicated, β will increases as tile sizes ts_2 through ts_d increases. ts_1 does not affect communication, since the outermost tile dimension is the dimension distributed among processors. On the other hand, increasing any tile size (ts) will increase the amount of computation per tile. Therefore, we expect that by appropriately selecting tile sizes, buffering factor greater than 2 is not necessary. It is also known that the improvement due to overlapping communication with computation is at most a factor of 2 [92].

Changing the tile size may alter other behaviors of the program, such as locality. However, the block cyclic distribution we use can control locality. Increasing the block size increases the number of tiles performed between communication are increased, and hence increases the amount of computation without affecting locality. Because block size does not affect the size of tile facet being communicated, the volume of communication stays unchanged. Hence, block sizes can be used to provide enough overlap with communication and computation without destroying locality. This does not change the legality of our approach, as multiplying ts_1 by block size gives equivalent behavior aside from the order of computation within a tile (or tile blocks.)

7.4.4 Code Generation

Now, let us discuss the code generation to realize the above placement of communications. Placement of the send is unchanged from the naïve placement, where values are sent after computing values of a tile. The

placement of the receive seems more complicated, since $\texttt{prev}(ti)$ may be an empty tile that we would like to avoid visiting. Since the wave-front hyper-plane may be growing (or shrinking) in size as the wave-front time step proceeds, both $\texttt{prev}(ti)$ and $\texttt{next}(ti)$ have a chance of not being in the outset. However, we must visit $\texttt{prev}(ti)$ even if there is no computation to be performed just to receive data for next time step.

We take advantage of the structure of loop nests after D-Tiling to efficiently iterate the necessary tiles. D-Tiling produces d loops to visit tile origins. The outermost loop iterates over time, and then the tile origins up to $d-1$-th dimension are visited. Then the final tile origin is computed using Equation 7.2. Furthermore, the check against the outset to avoid unnecessary point loops are after computing the last tile origin. Therefore, it is easy to visit some of the empty tiles by modifying the guards on outset.

Let $\texttt{InOutSet}(ti)$ be a function that returns true if the tile origin, ti is within the outset. Using the above, and previously defined function `sender`, `receiver`, and `next`, operations in a tile are guarded by the following:

- Computation of ti is performed if $\texttt{InOutSet}(ti)$ is true.
- Tile ti sends its values if $\texttt{InOutSet}(ti) \wedge \texttt{InOutSet}(\texttt{receiver}(ti))$ is true.
- Tile ti receives values if $\texttt{InOutSet}(\texttt{next}(ti)) \wedge \texttt{InOutSet}(\texttt{sender}(\texttt{next}(ti)))$ is true.

Note that for two tiles ti, and ti', the following equations hold:

$$ti = \texttt{sender}(\texttt{next}(ti'))$$

$$\texttt{receiver}(ti) = \texttt{next}(ti')$$

Thus, every send has a corresponding receive.

If a tile ti is in the OutSet, tile origins $[ti_1, ti_2, ..., ti_{d-1}]$ are visited *every* wave-front time step. When the ti_d is not in the outset for the current wave-front, the point-loops are skipped by the guard on OutSet. Therefore, if ti requires data at time t, then it is guaranteed that $[ti_1, ti_2, ..., ti_{d-1}]$ is visited at $t-1$ (or any other time step). Thus, we can check if $\texttt{next}(ti)$ requires data in the next time step, and receive even if the tile is not in the OutSet. The tile origins for $\texttt{next}(ti)$ can be computed by computing ti_d with $time+1$, and it does not matter if ti is not in the outset if the code is inserted outside the guard on OutSet for ti.

Figure 7.4 illustrates the structure of code we generate using the above strategy.

7.5 Evaluation

We evaluate our approach by applying our code generator to `PolyBench/C 3.2` [84]. Our evaluation is two-fold, we first show that despite the constraint on uniform dependences in one of the dimensions, most of `PolyBench` can be handled by our approach.

```
for (time = start; time <= end; time++)
   for (ti1=ti1LB+pid*ts1; ti1 <= ti1UB; ti1+=ts1*numP) {
      ...
         tid = getLastTI(time, ti1, ti2, ...)
         //issue receive for the next tile
         tidNext = getLastTI(time+1, ti1, ti2, ...)
         if (SenderInOutSet(ti1, ti2, ..., tidNext)
                && InOutSet(ti1, ti2, ..., tidNext)) {
            MPI_Irecv(&recvBuffer, ..., &recvReq];
         }
         //compute if in outset
         if (InOutSet(ti1, ti2, ..., tid)) {
            //point loops
            ...
            //send values
            if (ReceiverInOutSet(ti1, ti2, ..., tid)) {
                 MPI_Wait(&sendReq);
                //copy outputs to buffer
                ...
                MPI_Isend(&sendBuffer, ..., &sendReq);
            }
         }
         //values should have arrived while computing
         if (SenderInOutSet(ti1, ti2, ..., tidNext)
                && InOutSet(ti1, ti2, ..., tidNext)) {
            MPI_Wait(&recvReq);
            //copy received values to local memory
            ...
         }
   }
```

Figure 7.4: Structure of generated code. The function `getLastTI` is called for *time* and *time* + 1 to find `next`(ti). Then the issue of receive, computation, sends, and completion of receive is performed within the corresponding guards. Variables `recvReq` and `sendReq` are handles given by asynchronous communication calls that are later waited on.

We then compare the performance of our code with shared memory parallelization using PLuTo, but we expect to scale well beyond the number of cores on a single shared memory node. Our goal is to show comparable performance to PLuTo. We do not expect our code to scale any better than PLuTo as we use the same parallelization strategy based on tiling.

7.5.1 Applicability to `PolyBench`

Out of the 30 benchmarks in `PolyBench`, 23 satisfies the condition that at least one dimension of its tilable space is uniform. Among the remaining 7, 4 can be handled by splitting the kernel into multiple phases as discussed in Chapter 5. These benchmarks are `correlation`, `covariance`, `3mm`, and `ludcmp`. Although phase detection of these kernels is trivial, the general problem is difficult, and we do not address this problem. We handle these benchmarks by manually splitting into phases, and then combining the codes generated individually.

The three benchmarks that cannot be handled are the following:

- `cholesky` has transpose-like dependences resulting from the particular implementation of `PolyBench`. Cholesky decomposition can be handled if coded differently.
- `durbin`, and `gramschmidt` do not have large enough tilable bands to benefit from tiling based parallelism. These two kernels are not parallelized by PLuTo.

Even though we require uniform dependence on a dimension, we can handle virtually all of the `PolyBench`, since those that we cannot handle are not tilable either. We also note that all of the 23 benchmarks, as well as the different phases of the 4 that require phase detection, are all fully uniform after our uniformization step.

7.5.2 Performance Evaluation

We evaluate our approach on a subset of kernels in `PolyBench`. We exclude a number of kernels for the following reasons:

- `trmm`, `dynprog`, and `adi` have bugs and do not correctly implement their computation.
- `doitgen`, `reg_detect`, and `fdtd-apml` use single assignment memory, and thus no meaningful comparison can be made with other tools that retain original memory allocation.
- `symm`, and `ludcmp` use scalars for accumulation preventing PLuTo from parallelizing the computation.
- `atax`, `bicg`, `gemver`, `gesummv`, `mvt`, `trisolv`, and `jacobi-1d-imper` are small computations (only 2D loop,) and only run for less than a second sequentially with large problem sizes.

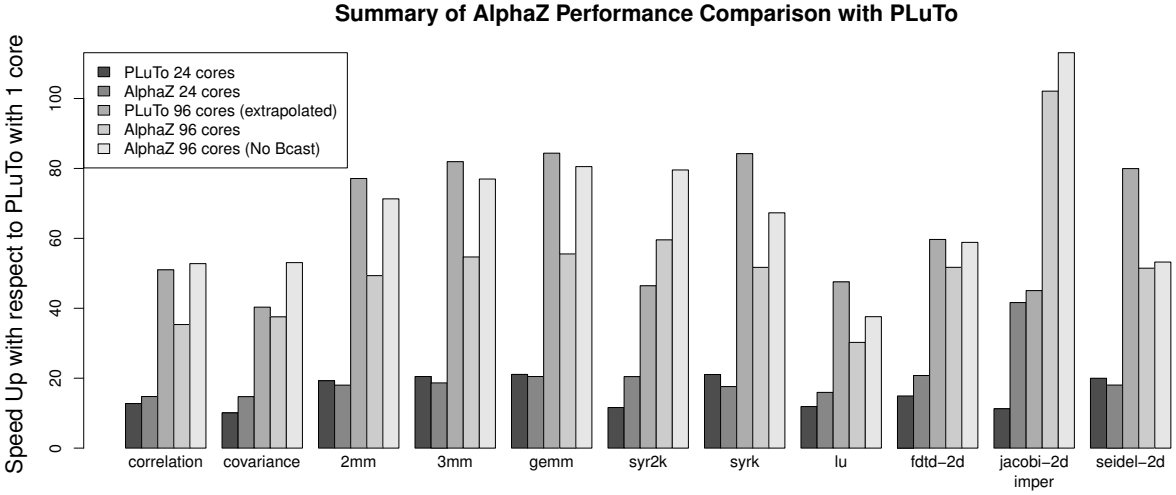


Figure 7.5: Summary of performance of our code generator in comparison with PLuTo. Numbers for PLuTo with 96 cores are extrapolated (multiplied by 4) from the speed up with 24 cores. MPIC (No Bcast) are number with the time to broadcast inputs removed. Broadcast of inputs takes no more than 2.5 seconds, but have strong impact on performance for the problem sizes used. This is a manifestation of the well known Amdahl's Law, and is irrelevant to weak scaling that we focus on. With the cost of broadcast removed, our code generator matches or exceeds the scaling of shared memory parallelization by PLuTo.

- `floyd-warshall` has very limited parallelism, and shows poor parallel performance (about 2x) with 24 cores when parallelized for shared memory via PLuTo.

This leaves us with the following 11 benchmarks: `correlation`, `covariance`, `2mm`, `3mm`, `gemm`, `syr2k`, `syrk`, `lu`, `fdtd-2d`, `jacobi-2d-imper`, and `seidel-2d`.

We measured the performance using Cray XT6m. A node in the Cray XT6m has two 12 core Opteron processors, and 32GB of memory. We used `CrayCC/5.04` with `-O3` option on the Cray. PLuTo was used with options `--tile --parallel --noprevector`, since prevector targets `ICC`. The problem sizes were chosen such that PLuTo parallelized code with 24 cores run for around 20 seconds.

The tile sizes were selected through non-exhaustive search guided by educated guesses. The tile sizes explored for PLuTo (and for our code up to 24 cores) were limited to square/cubic tiles, where the same tile size is used in each dimension. For our code executing on larger core counts, tile sizes were more carefully chosen. In most benchmarks, we found that the best tile sizes are different depending on the number of cores used. Note that PLuTo can only generate tiled code with fixed tile sizes.

The comparison is summarized in Figure 7.5. The results show that the MPI code produced by our code generator show comparable scaling as the shared memory parallelization by PLuTo. We require that the cost of initial broadcast to be removed for comparable scaling, which is not necessary when much larger problem sizes are used. In the following, we present the performance of each benchmark in more detail.

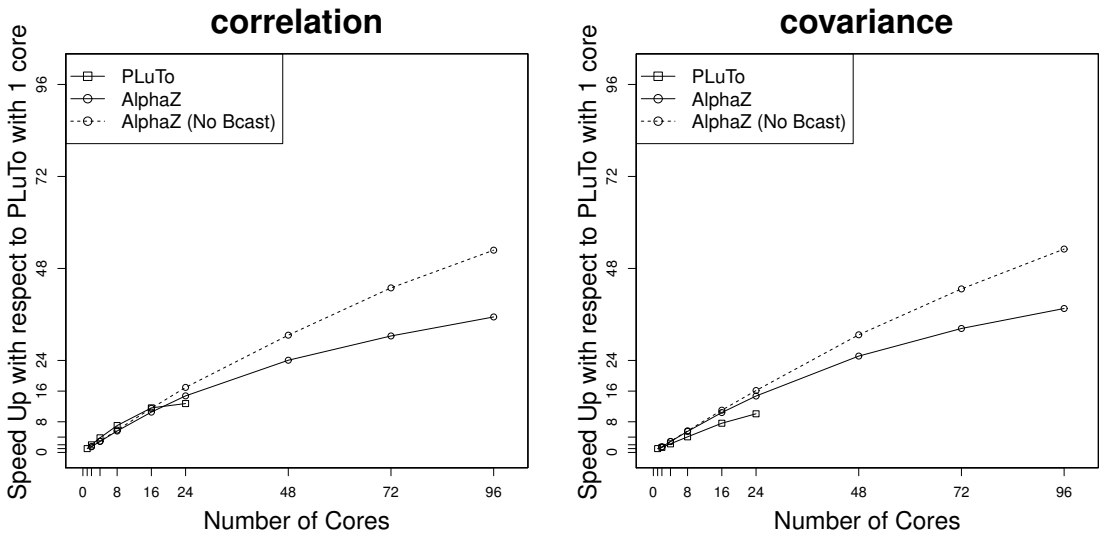


Figure 7.6: Performance of data mining kernels in `PolyBench`. Our MPI parallelization gives comparable or better performance in both benchmarks.

7.5.2.1 Data Mining

Figure 7.6 shows the results for `correlation` and `covariance`, categories as data mining in `PolyBench`. Both of these benchmarks have two phases, where the first phase normalizes the input data matrix using means and standard deviations. The normalization phase is $O(N^2)$ where N is the size of the input square matrix, while the latter phase is $O(N^3)$. Thus, we only parallelize the latter phase, and the normalization is only performed sequentially.

The performance of MPI parallelization is comparable to shared memory parallelization by PLuTo, and continues to scale beyond 24 cores. Our code performs better in `covariance`, which is due to a slightly different scheduling decisions by PLuTo and ISL schedulers. The default fusing strategy by PLuTo do not fuse a number of loop nests in `covariance`, and produces a sequence of 7 parallel loop nests. Using the same schedule to generate shared memory parallel code yields similar performance up to 24 cores.

7.5.2.2 Linear Algebra

Figure 7.7 illustrates the results for benchmarks in `PolyBench` categorized under `linear-algebra kernels` and `linear-algebra solvers`.

We require `3mm`, which is three matrix multiplications, where outputs of the two are used in the third, to be separated into two phases. The former phase is a single matrix multiplication, and the latter is essentially `2mm`. In fact, such separation of phases are performed by PLuTo scheduling as the outermost constant dimension. Thus, the result of a scheduling is already two tilable loop nests, and no special phase detection is required.

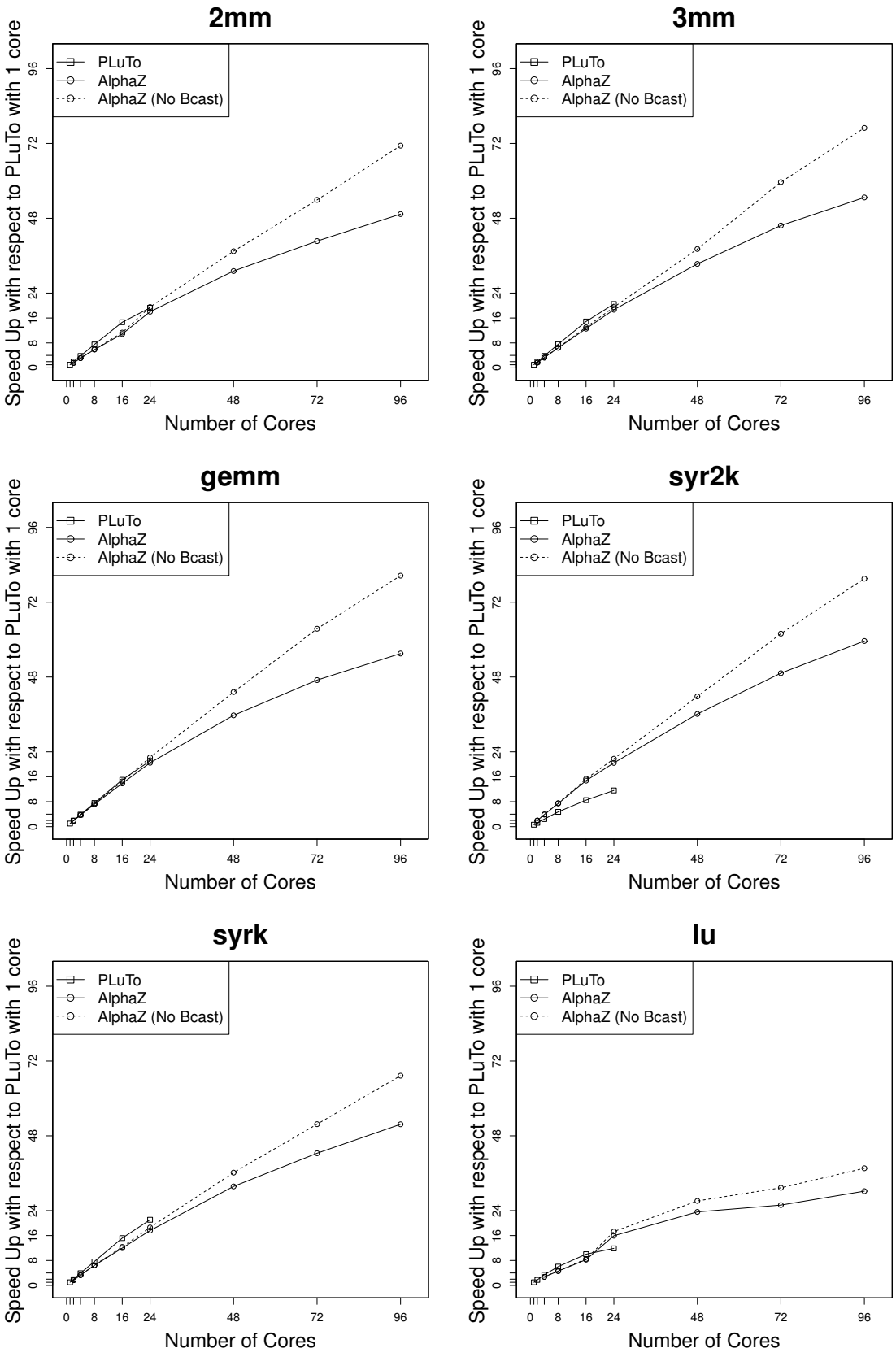


Figure 7.7: Performance of linear algebra kernels in `PolyBench`.

Most of these benchmarks do not require any communication after PLuTo scheduling, and show good scaling. We perform worse on some of the benchmarks, due to the quality of our sequential code. The code generated by AlphaZ is slightly slower in sequential execution when compared with PLuTo, and the difference gets amplified as number of cores increases. The performance difference comes from a mixture of multiple factors such as the overhead of parametric tiling, influence on compiler optimizations due to parametric tiles, and difference in memory allocations.

We perform better than PLuTo in `syr2k` due to the same reason as `covariance`. PLuTo does not fuse the loop nests, leading to two parallel loops. The performance of `lu` is notably worse than others. We use the same parallelization strategy as PLuTo, and we do not expect our code generator to scale better than PLuTo. Note that the extrapolation used in Figure 7.5 is a simple linear interpolation using the speedup at 24 cores, and assumes that the parallel efficiency stays the same at 96 cores. This is not the case with `lu`, and this is why the extrapolated performance of PLuTo appears to good in Figure 7.5.

7.5.2.3 Stencil Computations

Figure 7.8 depicts the performance of our code generator for stencil computations in `PolyBench`.

We outperform PLuTo in `fdtd-2d` and `jacobi-2d-imper` due to two different reasons. The skewing transformation for `fdtd-2d` is different because of memory-based dependences that PLuTo respects, where we do not. The skewing chosen by PLuTo results in more complex control structure, and negatively impacts performance.

In `jacobi-2d-imper`, values computed a time step is explicitly copied to another array before the next time step. Jacobi stencil in `PolyBench` uses explicit copying over other methods (e.g., pointer swaps) so that the computation can be analyzed via polyhedral analysis. (`jacobi-2d-imper` is the 2D data version of the code we used in Section 2.1.12.) UOV-based allocation[1] after PLuTo scheduling allocates a scalar for a statement that is immediately copied to an array allocated for the other statement. Thus, our code uses only one array as opposed to two used by shared memory parallelization using PLuTo.

We are significantly outperformed by PLuTo with `seidel-2d`. This is due to a combination of multiple factors. The best tile size we found for PLuTo in this benchmark was $8 \times 8 \times 8$. This is a surprisingly small tile considering the cache size and memory foot print. We found that the Cray compiler uses optimization that are otherwise unused when the tile sizes are small, compile time constant for this code. For instance, turning off loop unrolling has no influence on other tile sizes, but slows down the 8 cubic tiled program by more than 15%.

[1]Technically speaking, we use an obvious optimization of only allocating a scalar if the only dimension that carries any dependences is the inner-most statement ordering dimension.

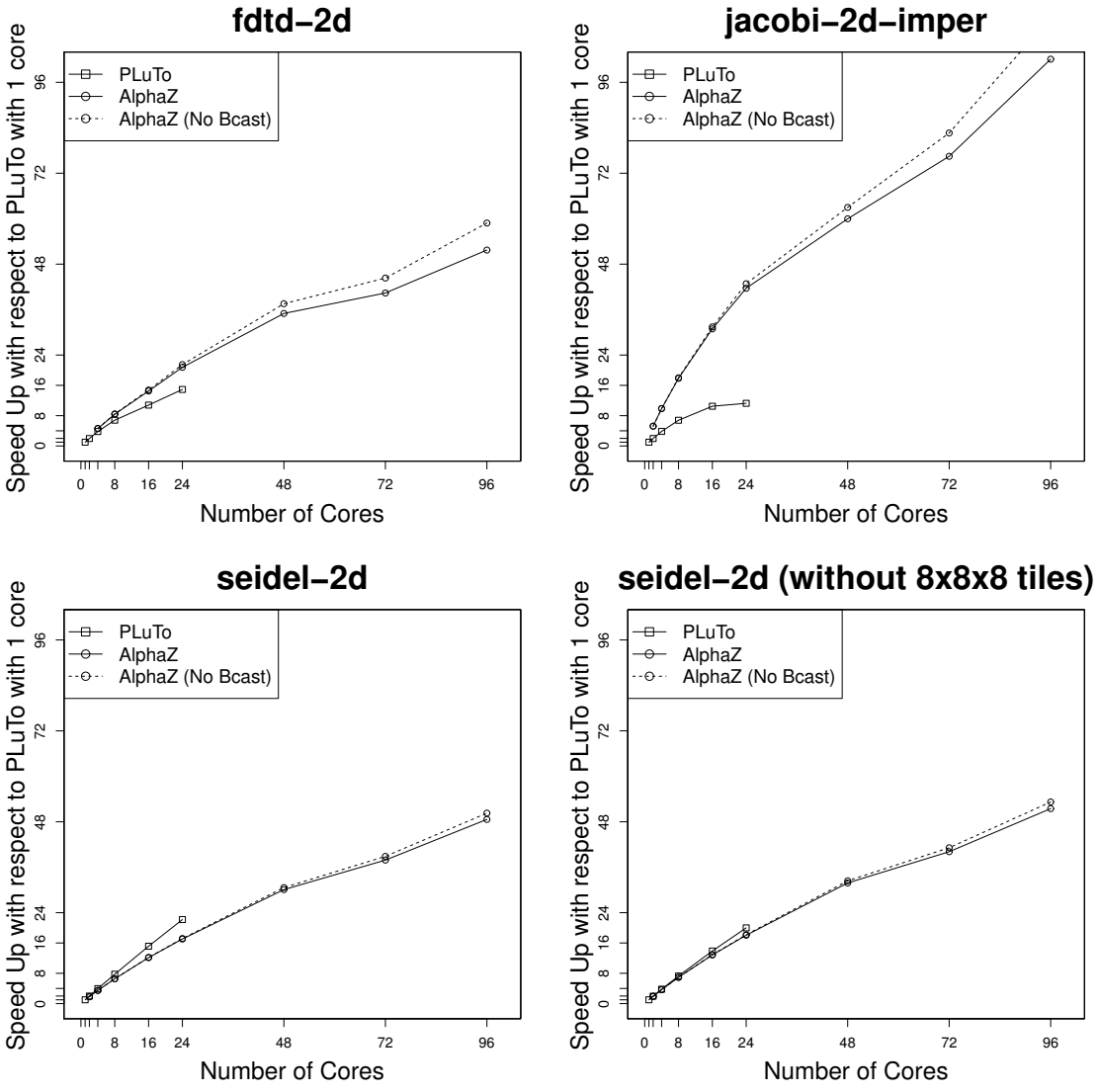


Figure 7.8: Performance of stencil kernels in `PolyBench`.

In addition to the lost compiler optimization due to parametric tiling, the dependence patterns in Gauss Seidel stencil computation negatively impact the performance of AlphaZ generated code. The Gauss Seidel in `PolyBench` does not update boundary values of the 2D data array, and only updates the interior points at each time step. Therefore, the program has only one statement, and boundary cases are all handled implicitly through loop bounds and memory. However, when memory based dependences are ignored, a number of boundary conditions arise as different statements. The number of statements in AlphaZ generated code reaches 14, where 13 of them are boundary cases, and comes with non-negligible control overhead.

We are significantly outperformed as a combination of these reasons. We do still have linear scaling, and as shown in Figure 7.8, the scaling is very similar to PLuTo once small tile sizes that enable additional compiler optimizations are removed from the data set.

7.6 Summary and Discussion

In this chapter, we have presented our approach for generating distributed memory parallelization. The key idea in our approach is to only allow uniform dependences to cross processor boundaries. Although this may seem restrictive, effectively all benchmarks in `PolyBench` that can be efficiently parallelized by wave-front execution of tiles can be handled. Because of many simplifications that can be made thanks to uniform dependences, communication becomes extremely simple, and resulting codes have similar scaling as shared memory parallelization.

Uniformization (or localization) was once a popular approach for parallelization. With the increased popularity of shared memory architectures, which makes handling of affine dependences much easier, techniques for uniformization are now rarely used. However, when communication can no longer be made implicit via shared memory, resurrecting uniformization is a possible approach.

There are a number of parameters that can be explored in this parallelization scheme. The current strategy for uniformization is to uniformize as much as possible, and then check if there is at least one dimension that is uniform. There is really no need for intra-processor dependences to be uniformized, and hence how to select dependences to uniformize posed an interesting challenge.

Moreover, the memory allocation of the original program can also be reused if desired. The original allocation is also a legal memory allocation with parametric tiling, provided that the memory-based dependences are taken into account. Thus, one can envision partially respecting memory-based dependences, while finding new memory allocations for others. As illustrated by `jacobi-2d-imper` benchmark, re-allocating memory can significantly improve performance. However, the original allocation is, in many cases, efficient, and re-allocating may result in performance degradation. Such trade-off was not explored in our work.

It is also possible to use both MPI and OpenMP so that intra-node parallelism is exploited by OpenMP. Since the MPI implementation is very efficient on shared memory, we did not explore such hybrid approach.

However, it is a trivial extension to our work to generate hybrid version; one can simply apply multi-level tiling, and parallelize the inner tiles with OpenMP. Multi-level tiling can also be used for other levels of memory hierarchy; registers, different level of caches, TLB, and so on. This is another large space that remains to be explored.

Chapter 8

Polyhedral X10

Because parallelism has gone mainstream, the problem of improving programmer productivity is now increasingly important. It was the goal of the DARPA HPCS program, initially for the supercomputing niche, but is no longer a niche problem. A number of new parallel programming languages are being actively developed and explored [7, 17, 80, 105, 116, 130]. These languages all employ new programming models to ease the parallel programming effort.

While designing and writing parallel programs is significantly harder than its sequential counterpart, one problem that is even harder is debugging. This is due to the non-deterministic nature of parallel execution, and the accompanying difficulty of reproducing errors. Therefore the problem of static analysis of parallel programs is becoming critical. Some recent parallel languages also attempt to ease the debugging effort. For example, Titanium [130] can check for possible deadlocks at compile time. Most parallel constructs in X10 [105] are designed such that no (logical) deadlocks occur.

In this chapter, we present static race detection for a subset of X10 programs. By providing race-free guarantee, we can prove determinacy of a program. The ability to detect races statically will allow programmers to correct them as they write the program, greatly improving their productivity. The subset of X10 we consider includes its core parallel constructs, `async` and `finish`. We also require that the loop bounds and array accesses to be affine to fit the *polyhedral model*, a mathematical framework for reasoning about program transformations, used mainly in automatic parallelization. We improve upon previous techniques for static data race detection in two key directions:

- Our analysis is statement *instance-wise* and array *element-wise*. Most existing approaches analyze race of static statements and conservatively flags as race if two statements that may happen in parallel access the same variable. Our analysis will find sets of statement *instances* (i.e., statement executed when loop counters take specific values) that may happen in parallel, and only flags as race if they access the same *element* of an array.

- In comparison with other methods that support both instance-wise and element-wise analysis [14, 23], our work supports parallelism based on finish/async constructs, which are more expressive than *doall* type parallelism considered in prior work.

Consider the following code fragment that uses the `async` construct of X10.

```
for (i in 0..N) {
   S0
   async S1
}
```

The above code sequentially executes `S0` at each iteration of the loop, and after executing `S0`, spawns a new activity to execute an instance of `S1`. This activity executing `S1` may run in parallel with `S0` in the *next* iteration. Such parallelism cannot be expressed as *doall* type parallelism, and no clean way to express such parallelism in the polyhedral model is available. The focus of polyhedral techniques has been automatic parallelization with *doall*, since it is the most common form of expressing parallelism in many parallel programming models.

Our race detection is based on an adaptation of array dataflow analysis [30] from the polyhedral model. This allows us to have the same level of precisions as other work based on the polyhedral model, but is applicable to finish/async based parallelism.

Specifically, our key contributions we present in this chapter are:

- Characterization of the HB relation as an *incomplete lexicographical order*. This is the key to reuse techniques from the polyhedral model for X10 programs.
- Adaptation of Array Dataflow Analysis [30] for X10 programs. Array dataflow analysis answers the question, which instance of which statement produced the value being used for each statement instance. We have extended this analysis to handle X10 programs.
- Race Detection using array dataflow results. The key idea is that once we have solved the dataflow problem, we can identify the set of instances that cause a race by pinpointing the set of array cells which have multiple producers.
- Prototype implementation of a verifier for our subset of X10. If a program has races, our tool can tell precisely which statement instances are involved.

Acknowledgements

The material in this chapter is developed in tight collaboration with Dr. Paul Feautrier and Dr. Vijay Saraswat. In particular, we would not have had the elegant formulation of dataflow analysis for X10 programs without Paul. Vijay has developed operational semantics for the subset of the X10 we handle, and connected it to the definition of Happens Before relationship we define in Section 8.1.2, adding much strength to our claims.

8.1 A Subset of X10

Our main interest is in intra-procedural analysis, and we wish to address the integration of finish/async concurrency with loops over array based data-structures in a pure form. Hence we will only consider assignment statements, sequencing, finish, async and for loops, and variables that range over integers and arrays of integers. This allows us to state certain properties of the key relations—Happens Before and its closely related May Happens in Parallel—in a pure form. Our formal treatment can be extended *mutatis mutandis* with conditionals, local variables, potentially infinite loops, method calls, objects and functions etc., but we restrain ourselves for reasons of space.

The subset of X10 [105] we consider consists of the following control constructs:

- Sequence ($\{ST\}$): Composes two statements in sequence.
- Sequential `for` loop: We assume all loops have an associated loop iterator. X10 loops may scan a multidimensional iteration space. However, we assume that such loops have been expanded into a nest of unidimensional loops.
- Parallel activation, `async`: The body of the `async` is executed by an independent lightweight thread, called `activity` in X10.
- Termination detection, `finish`: An activity waits for all activities spawned within the body of `finish` to terminate before proceeding further. In addition, each program has an implicit `finish` as its top level construct.

We also require the program to fit the polyhedral model. The polyhedral model requires loop bounds, and array access to be affine expressions of the surrounding loop indices. Multidimensional arrays in X10 and Java are in fact trees of one-dimensional arrays. As such, they support many operations beyond simple subscripting. An example is row interchange in constant time. Detection of such uses is beyond the scope of this paper; see for instance the work by Wu et al. [126] for an abstract interpretation approach.

Note that the full language permits some additional constructs: for instance the (conditional) atomic block (`when(c) S`) . This construct permits data-dependent synchronization in general and barrier-style *clocks* [106] in particular. The `at` construct permits computation across multiple places. We leave the integration of these statements into the analysis presented in this chapter as future work.

8.1.1 Operational Semantics

We provide a simple, concise structural operational semantics (SOS) for the fragment of X10 considered in this paper. This semantics is considerably simpler than [106] because it eschews the "Middleweight Java" approach in favor of directly specifying semantics on statements. Unlike [66] there is no need to translate

the statement to be executed into different kinds of tree-like structure; the information is already contained in the lexical structure of the statement and can be elegantly exploited using SOS based structural rules.

In this section, we present the semantics, characterize certain syntactic properties of statements (the happens before and the may happens in parallel relation), and relate them to behavioral properties. For simplicity of exposition, we chose to use a sequentially consistent memory model. In future work we expect to apply the methods of [107] to adapt the analysis techniques developed in this paper to relaxed memory models.

We assume that a set of (typed) locations `Loc`, and a set of values, `Val`, is given. `Loc` typically includes the set of variables in the program under consideration. With every d-dimensional $N_1 \times \ldots \times N_d$ array-valued variable `a` of type array are associated a set of distinct locations, designated `a(0,...,0), ..., a(N`$_1$`-1, ..., N`$_d$`-1)`. The set of values includes integers and arrays.

A *heap* is a partial (finite) mapping from `Loc` to `Val`. For h a heap, l a location and v a value by $h[l = v]$ we shall mean the heap that is the same as h except that it takes on the value v at l. By $h(l)$ we mean the value e to which h maps l.

Definition 8.1 (Expressions) *We assume that a set of* RHS expressions *(ranged over by $e, e', e_0, e_1, \ldots$) that denote values is defined. RHS expression include variables (e.g., `x`), literals (e.g., `0`), array accesses (e.g., `a(p)`), and appropriate operations over integers (e.g., addition). We also assume that a set of* LHS expressions *(ranged over by $a, a', a_0, a_1, \ldots$) that denote locations is defined. These include variables and array accesses.*[1] *We extend h to a map from RHS expressions to values and from LHS expressions to locations in the obvious way.*

Definition 8.2 (Statements) *The statements are defined by the productions:*

(Statements) S, T	$::=$
$a = e$;	Assignment.
$\{S\,T\}$	Execute S then T.
`for`$(x$ `in` $e1..e2)\ S$	Execute S for x in $e1 \ldots e2$.
`async` S	Spawn S.
`finish` S	Execute S and wait for termination.

The set of paths $\mathcal{P}[\![S]\!]$ corresponding to a statement S is given as follows. For a set of paths U, we let $x\,U$ stand for the set of paths $x\,s$, for $s \in U$.

$$\begin{aligned}
\mathcal{P}[\![a = e]\!] &= \{\epsilon\} \\
\mathcal{P}[\![\{S\,T\}]\!] &= \{\epsilon\} \cup \mathtt{0}\mathcal{P}[\![S]\!] \cup \mathtt{1}\mathcal{P}[\![T]\!] \\
\mathcal{P}[\![\mathtt{for}(x\ \mathtt{in}\ e1..e2)\ S]\!] &= \{\epsilon\} \cup \mathtt{x}\mathcal{P}[\![S]\!] \\
\mathcal{P}[\![\mathtt{async}\ S]\!] &= \{\epsilon\} \cup \mathtt{a}\mathcal{P}[\![S]\!] \\
\mathcal{P}[\![\mathtt{finish}\ S]\!] &= \{\epsilon\} \cup \mathtt{f}\mathcal{P}[\![S]\!]
\end{aligned}$$

[1] Thus, as is conventional in modern imperative languages, the notation `a`(i) is ambiguous. When used on the LHS it represents a location and when used on the RHS it represents the contents of that location.

The statements for which the operational semantics is defined are assumed to satisfy some static semantic conditions (e.g., well-typedness.) We omit the details. Note that in a `for` loop the index variable is considered bound. To avoid dealing with alpha renaming, we assume that in the statement under consideration no two loop index variables are the same. Also note that the set of paths for a statement is non-empty and prefix-closed, hence defines a tree. A path $p \in \mathcal{P}[\![S]\!]$ is *terminal* if it is not a proper prefix of any other in $\mathcal{P}[\![S]\!]$.

The following proposition follows from the definition of statements and its paths, where the only place of divergence is sequential composition.

Proposition 8.3 *For a statement S, let $s\,x$ and $s\,y$ be two distinct paths in $\mathcal{P}[\![S]\!]$. Then either $x = 0 \wedge y = 1$ or $x = 1 \wedge y = 0$.*

However, the paths in $\mathcal{P}[\![S]\!]$ are *static*, where an element involving loop iterators are considered equal if the names of the iterators match. We define an *instance* of a path as $t = \theta(s)$, where θ is a substitution applied to $s \in \mathcal{P}[\![S]\!]$ mapping index variables to integers.[2] Given S, $s \in \mathcal{P}[\![S]\!]$, and an instance of $t = \theta(s)$ note that θ can be recovered uniquely.

Another version of Proposition 8.3 can be made for path instances, following the observation that a common prefix s of two paths in $\mathcal{P}[\![S]\!]$, may no longer in common only as a result of the application of θ, which only affects index variables.

Proposition 8.4 *For a statement S, let $s\,x$ and $s\,y$ be two distinct instances of paths in $\mathcal{P}[\![S]\!]$. Then either $x < y$ or $y < x$.*

We also introduce a notation to denote sub-statements. Let S be a statement and $s \in \mathcal{P}[\![S]\!]$. We use the notation $S\hat{}s$ to refer to the sub-statement of S obtained by traversing the path s from the root. Given a path instance $t = \theta(s)$, we define $S\hat{}t$ to be $\theta(S\hat{}s)$, that is, θ applied to the statement obtained by traversing the path s from the root of S. This definition is justified by the fact that θ is unique for each path instance.

Definition 8.5 (Read and write set) *Let S be a statement, and $s \in \mathcal{P}[\![S]\!]$ a terminal path (or path instance). We let $rd(S, s)$ denote the set of locations read by $S\hat{}s$ and $wr(S, s)$ the set of locations written in $S\hat{}s$.*

Let s, t be two paths or path instances for S. We say s write-affects *t if $wr(S, s) \cap (rd(S, t) \cup wr(S, t))$ is non-empty. We say that s and t* conflict *if s write-affects t or* vice versa. *We say that s* self-conflicts *if $rd(S, t) \cap wr(S, t)$ is non-empty.*

For instance, let S be the statement `for`$(i$ `in` $0..10)$ $a(i) = a(i) + 1$. Then the path $[\epsilon]$ self-conflicts, as does

[2] Usually, we will be concerned only with path instances that satisfy the bounds conditions for the index variable.

[i]. But the path (instance) [0] does not. In fact the paths [i],[j] do not conflict if i, j are distinct integers (in the given range).

Note that $S\hat{}s$ may be a statement with free variables (e.g., parameters), hence the set of locations read/written may be symbolic (i.e., heap-dependent at run-time.)

Execution relation. As is conventional in SOS, we shall take a *configuration* to be a pair $\langle S, h\rangle$ (representing a state in which S has to be executed in the heap h) or h (representing a terminated computation.)

The operational execution relation $\longrightarrow$ is defined as a binary relation on configurations.

The axioms and rules of inference are:

$$\frac{l = h(a), v = h(e)}{\langle a = e, h\rangle \longrightarrow h[l = v]} \tag{8.1}$$

$$\frac{\langle S, h\rangle \longrightarrow \langle S', h'\rangle \mid h'}{\begin{array}{rcl} \langle \{S\,T\}, h\rangle & \longrightarrow & \langle \{S'\,T\}, h'\rangle \mid \langle T, h'\rangle \\ \langle \texttt{async}\ S, h\rangle & \longrightarrow & \langle \texttt{async}\ S', h'\rangle \mid h' \\ \langle \texttt{finish}\ S, h\rangle & \longrightarrow & \langle \texttt{finish}\ S', h'\rangle \mid h' \\ \langle \{\texttt{async}\ T\,S\}, h\rangle & \longrightarrow & \langle \{\texttt{async}\ T\,S'\}, h'\rangle \mid \langle \texttt{async}\ T, h'\rangle \end{array}} \tag{8.2}$$

One can think of these rules as propagating an "active" tag from a statement to its constituent statements. The first rule says that if $\{S\,T\}$ is active then so is S (that is, any transition taken by S can be transformed into a transition of $\{S\,T\}$.) The second rule says that if `async` S is active, then so is S. The third rule says the same thing for `finish` S. The fourth rule captures the essence of `async` (we call it the *"out of order"* rule). It says that in a sequential composition $\{$`async` $T\,S\}$, the second component S is also active. Thus one can think of `async` S as licensing the activation of the following statement (in addition to activating S).

The first `for` rule terminates execution of the `for` statement if its lower bound is greater than its upper bound.

$$\frac{l = h(e_0), u = h(e_1), l > u}{\langle \texttt{for}(x\ \texttt{in}\ e_0..e_1)\ S, h\rangle \longrightarrow h} \tag{8.3}$$

The recursive rule performs a "one step" unfolding of the `for` loop. Note that the binding of x to a value l is represented by applying the substitution $\theta = x \mapsto l$ to S, rather than by adding the binding to the heap. This is permissible because x does not represent a mutable location in S.

$$\frac{\begin{array}{l} l = h(e_0), u = h(e_1), l \leq u, m = l + 1, T = S[l/x] \\ \langle T, h\rangle \longrightarrow \langle T', h'\rangle \mid h' \end{array}}{\begin{array}{l} \langle \texttt{for}(x\ \texttt{in}\ e_0..e_1)\ S, h\rangle \longrightarrow \\ \qquad \langle \{T'\ \texttt{for}(x\ \texttt{in}\ m..u)\ S\}, h'\rangle \mid \langle \texttt{for}(x\ \texttt{in}\ m..u)\ S, h'\rangle \end{array}} \tag{8.4}$$

We now define appropriate semantical notions.

Definition 8.6 (Semantics) *Let $\xrightarrow{\star}$ represent the reflexive, transitive closure of $\longrightarrow$. The operational semantics, $\mathcal{O}[\![S]\!]$ of a statement S is the relation*

$$\mathcal{O}[\![S]\!] \stackrel{def}{=} \{(h, h') \mid \langle S, h\rangle \xrightarrow{\star} h'\}$$

Sometimes a set of visible *variables is defined by the programmer, with a refined notion of semantics:*

$$\mathcal{O}[\![S, V]\!] \stackrel{def}{=} \{(h, h'\big|_V) \mid \langle S, h\rangle \xrightarrow{\star} h'\}$$

where for a function $f : D \to R$ and $V \subseteq D$ by $f\big|_V$ we mean the function f restricted to the domain V.

Note in the above definition we have chosen not to restrict the set of variables over which the input heap is defined. In a more complete treatment of the semantics, we would introduce the `new` operation which permits dynamic allocation of memory, and define the program as being executed in a heap that is initially defined over only the input array of strings containing the command line arguments.

Definition 8.7 (Determinacy) *A statement S with set of observables V is said to be* scheduler determinate over V *(or just* determinate *for short) if $\mathcal{O}[\![S, V]\!]$ represents the graph of a function.*

S is said to be scheduler determinate *if it is scheduler determinate over the set of its free variables.*

8.1.2 Happens Before and May Happen in Parallel relations

We now establish two structural relations on statements, and connect them to the dynamic behavior of the statements.

Definition 8.8 (Happens Before) *Given a statement S, two terminal paths q, r in $\mathcal{P}[\![S]\!]$, and instances of paths $i = \theta_q(q)$ and $j = \theta_r(r)$ we say that i* happens before *j, and write $i \prec j$, if for some arbitrary label sequences s, t, u, some sequence c over integers, and integers m_0, m_1 with $m_0 < m_1$:*

$$i = s\, m_0\, c \vee i = s\, m_0\, c\, \mathtt{f}\, t, \text{ and,}$$
$$j = s\, m_1\, u$$

The definition does not explicitly mention `async` nodes. The label `a` may occur in i, but only in s or in t. In the first case the occurrence can be ignored because we are considering two paths that lie within the same `async`. In the latter case the occurrence may be ignored because it is covered by a `finish`. The intuition is that the "asyncness" of a node can never cause it to happen before some other node. But the "finishness" of a node can—it suppresses all downstream asyncs. This intuition is formalized in the next section.

The following proposition is easy to establish by reasoning about sequences.

Proposition 8.9 *(Transitivity) If $i \prec j$ and $j \prec k$ then $i \prec k$. (Asymmetry) If $i \prec j$ then it is not the case that $j \prec i$. (Irreflexivity) For no i is it the case that $i \prec i$.*

Thus $\prec$ is a strict order. But it is not total. Consider $i = 0a$ and $j = 1$. It is not the case that $i \prec j$ or $j \prec i$.

Definition 8.10 (May Happen In Parallel) *Given a statement S, two terminal paths q, r in $\mathcal{P}[\![S]\!]$, and instances of paths $i = \theta_q(q)$ and $j = \theta_r(r)$ we say that i* can start with j running, *if for some arbitrary label sequences s, t, u, some sequence c over integers, and integers m_0, m_1 with $m_0 < m_1$:*

$$i = s\, m_0\, c\, \mathtt{a}\, t, \text{ and,}$$
$$j = s\, m_1\, u$$

We say that i may happen in parallel with j, *and write $i \# j$, if j starts with i running, or i starts with j running.*

Proposition 8.11 *Let S be a statement with two path instances i, j. Then $i \# j$ iff $\neg(i \prec j) \wedge \neg(j \prec i) \wedge i \neq j$.*

The proof of the forward direction is easy. In the backward direction we need Proposition 8.4.

Definition 8.12 (Race) *Given a statement S and two sub-statements T and U, we say that there is a race involving T and U, if for some legal instances $t = \theta_t(T)$ and $u = \theta_u(U)$, $t \# u$ and memory accesses by t and u conflict.*

8.1.3 Correspondence

We now establish the relationship between the HB and MHP relations and the transition relation. The formal language we are working with does not have conditionals, or local variables, or infinite loops. This means that every sub-statement will execute in every initial heap. Hence it is possible to characterize the MHP and HB relations in very simple terms.

The key idea in establishing the correspondence is to surface the path/time stamp of a statement in the transition relation. We label each step by the "reason" for the step—the path (from the root) to the substatement that triggers (is the base case for) the transition.

We proceed as follows. First we define *labeled statements*—each substatement is labeled with the path from the root. Next we label transitions. The rules are a straightforward adaptation of Rules 8.1—8.4. The only point worth noting is that in the recursive rule for `for`, the substitution $S[l/x]$ replaces x by l in the labels of all substatements of S as well.[3]

$$\frac{l = h(a), v = h(e)}{\langle a = e^s, h\rangle \longrightarrow^s h[l = v]} \tag{8.5}$$

$$\frac{\langle S, h\rangle \longrightarrow^s \langle S', h'\rangle \mid h'}{\begin{array}{rcl} \langle \{S\, T\}, h\rangle & \longrightarrow^s & \langle \{S'\, T\}, h'\rangle \mid \langle T, h'\rangle \\ \langle \{\mathtt{async}\ T\, S\}, h\rangle & \longrightarrow^s & \langle \{\mathtt{async}\ T\, S'\}, h'\rangle \mid \langle \mathtt{async}\ T, h'\rangle \\ \langle \mathtt{async}\ S, h\rangle & \longrightarrow^s & \langle \mathtt{async}\ S', h'\rangle \mid h' \\ \langle \mathtt{finish}\ S, h\rangle & \longrightarrow^s & \langle \mathtt{finish}\ S', h'\rangle \mid h' \end{array}} \tag{8.6}$$

[3] In Rule 8.7, s is the label for the whole `for` statement.

$$\frac{l = h(e_0), u = h(e_1), u > l}{\langle \texttt{for}(x \texttt{ in } e_0..e_1)\ S^s, h\rangle \longrightarrow^s h} \tag{8.7}$$

$$\frac{\begin{array}{l} l = h(e_0), u = h(e_1), l \leq u, m = l+1, T = S[l/x] \\ \langle T, h\rangle \quad \longrightarrow^s \quad \langle T', h'\rangle \mid h' \end{array}}{\begin{array}{l} \langle \texttt{for}(x \texttt{ in } e_0..e_1)\ S, h\rangle \quad \longrightarrow^s \\ \qquad \langle \{T' \texttt{ for}(x \texttt{ in } m..u)\ S\}, h'\rangle \mid \langle \texttt{for}(x \texttt{ in } m..u)\ S, h'\rangle \end{array}} \tag{8.8}$$

Clearly this transition system is conservative over the previous one—it merely decorates each step with extra information.

Theorem 8.13 (Characterization of HB) *Let S be a statement and q, r terminal paths in $\mathcal{P}[\![S]\!]$.*

If $q \prec r$ then for any heap h, in any labeled transition sequence starting from $\langle S, h\rangle$ containing q and r, (the transition labeled with) q occurs before (the transition labeled with) r.

(Converse) If for all heaps h and all transition sequences started from $\langle S, h\rangle$ containing q and r it is the case that q occurs before r then $q \prec r$.

The forward direction is proved by structural induction on S. The key case is sequential composition $U \equiv \{S\,T\}$ in which q is a path leading into S and r into T. Here, the only "out of order" transition possible is because of the "out of order" rule, which requires S be an `async`. But since $q \prec r$ we know that $q \equiv s0c$ or $q \equiv s0cft$ and $r \equiv s1u$ (with s being the label for U). Hence S cannot be an `async` since its type is specified by the first label of cft. In the converse direction, without loss of generality, let $q \equiv s0t$ and $r \equiv s1u$. If the first symbol in t that is not an integer is an `a`, then we show in the proof that the "out of order" rule can be used to construct an execution sequence in which r precedes q, contradicting our assumption. Hence $q \prec r$.

The proof of the following theorem is similar.

Theorem 8.14 (Characterization of MHP) *Let S be a statement and q, r terminal paths in $\mathcal{P}[\![S]\!]$.*

If $q\#r$ then for any heap h there is a transition sequence starting from $\langle S, h\rangle$ containing q and r s.t. q occurs before r and another such that r occurs before q.

(Converse) If for all heaps h there is a transition sequence starting from $\langle S, h\rangle$ containing q and r s.t. q occurs before r and another such that r occurs before q, then $q\#r$ or $r\#q$.

Proposition 8.15 *Let S be a statement. If no two sub-statements are in a race, then S is determinate.*

The converse is not true. There may be a race but it may be *benign* in the sense that it does not affect the outcome of the program. Consider:

```
finish {
  async x=1; // S0=[f0a]
  x=1;       // S1=[f1]
}
```

Statements S0 and S1 are in a race (they may happen in parallel and their write sets overlap,) however, the statement is determinate, it will always yield a heap with the only change from intial state being the mapping from `x` to `1`.

8.2 The "Happens-Before" Relation as an Incomplete Lexicographic Order

In this section, we formulate the "happens-before" relation, in a manner familiar from polyhedral analysis. In the polyhedral analysis of sequential languages, statement instances in a program are given unique time stamps represented as *integer vectors*. These vectors are ordered lexicographically—this order is sufficient to capture the idea of "happens before" for a sequential language.

The strict lexicographic order is defined for two distinct such integer vectors u and v as follows:

$$u \ll v \equiv \bigvee_{p \geq 0} u \ll_p v, \tag{8.9}$$

$$u \ll_p v \equiv \left(\bigwedge_{k=1}^{p} u_k = v_k \right) \wedge (u_{p+1} < v_{p+1}) \tag{8.10}$$

As we saw in Section 8.1.2, the happens before order in X10 must be sensitive to the presence of `finish` and `async` nodes. To take these constructions into account, we will use the paths of Section 8.1.2—vectors of integers, loop counters and the letters `a` and `f`—as time stamps. Polyhedral analyses can take loop counters symbolically, but reason about path instances, where loop counters take some integer value.

We will consider only terminal paths. The lexicographic order may be extended to paths simply by specifying how to order the additional symbols a and f, for instance by assuming that `a` < `f` and that they occur later than integers and loop counters. This convention is irrelevant, since, by Proposition 8.4, we will never have to compare `a` or `f` to any other item in a path provided, we only compare *distinct* vectors.

Given a time stamp q, $|q|$ is its dimension, $q_i, 1 \leq i \leq |q|$ (sometimes written $q[i]$) is its i-th component, and $q[i..j], i \leq j$ is the vector whose components are $q_i, q_{i+1}, \ldots q_j$. A common shorthand for $q[i..|q|]$ is $q[i..]$.

A time stamp in which the loop counters have been replaced by integers denotes at most one instance of an elementary statement or *operation*. The admissible values (legal θ) are constrained to be within the enclosing loop bounds, which are assumed to be affine. The set of admissible values for the time stamps of statement S, the *iteration domain* of S, is written $\mathcal{D}_S$. Under the above hypothesis, $\mathcal{D}_S$ is a polyhedron.

We now reconstruct the "happens-before" relation as a "relaxed" lexicographic order. We start from the trivial observation that:

$$\textbf{true} \equiv (q \ll r) \vee (q = r) \vee (r \ll q).$$

This suggests that $q \prec r$ be constructed as a case distinction:

- $q \ll r \rightarrow ?$

- $q = r \rightarrow$ **false**
- $r \ll q \rightarrow ?$

The case $q = r$ is obvious, since an operation cannot execute before itself. Let us now show that if $r \ll q$, then $q \prec r$ is impossible. In the notations of Definition 8.8, let s be the common prefix of q and r: $q = s.x.u$ and $r = s.y.v$. By Proposition 8.4, either $x < y$ or $y < x$ is true, and $r \ll q$ implies that $x > y$. Then, Definition 8.8 implies that $q \prec r$ cannot be true.

The conclusion is that in the above disjunction, only the first case has to be considered. This in turn can be expanded according to the definition (8.9) of $\ll$:

- $q \ll_0 r \rightarrow ?$
- $q \ll_1 r \rightarrow ?$
- ...
- $q \ll_n r \rightarrow ?$

The case distinction extends until $q \ll_k r$ is obviously false, i.e., when q_k and r_k are different integers, since all predicates $q \ll_{k'} r, k' \geq k$ contains the constraint $q_k = r_k$.

Let us now consider one of the cases $q \ll_p r$. The two time stamps have a common prefix $q[1..k] = r[1..k]$, and by the same reasoning as above, $q_{k+1} = 0$ and $r_{k+1} = 1$. We are then in a position to apply Definition 8.8. If the first letter in the vector $q[k+1..]$ is an `f` or if there is no letter, then $q \prec r$ is true, and otherwise is false.

The discussion above can be summarized by the following algorithm:

Here, h denotes a disjunction of affine constraints, which is initialized to **false**, is augmented each time line (2c) is executed, and is the "happens-before" predicate when the algorithm terminates.

In what follows, in the interest of compactness, we will allow sequences with more that two items, and timestamps containing integers larger than 1.

Example

Let us apply Algorithm H to the example shown in Figure 8.1. The time stamps associated with each statement are as follows: `S0`: $[0, \mathtt{f}, 0, i, \mathtt{a}]$, `S1`: $[0, \mathtt{f}, 1, j]$, and `S2`: $[1]$.

We first ask if $S0 \prec S1$. Then $q = [0, \mathtt{f}, 0, i, \mathtt{a}]$ and $r = [0, \mathtt{f}, 1, j]$.

- We start from $k = |q| = 5$, $b =$ true, and $h =$ false.
- At $k = 5$, b becomes false, since $q_5 = \mathtt{a}$.
- Since q_k does not point to an `f` until $p = 1$, no changes occur. At $k = 1$, $q \ll_0 r \equiv q_1 < r_1$ is false, and hence $S0$ does not happen before $S1$.

Algorithm 8.1 Algorithm H

Input:

Two paths q, r.

Output:

The constraint h in the loop counters of q, r (if any) which captures the precise conditions under which $q \prec r$.

Algorithm:

1. Initialization

   $h := \emptyset$
   $b :=$ **true**

2. **for** $k = |q|$ downto 1:

   (a) **if** $q_k =$ `a` **then** $b :=$ **false**
   (b) **if** $q_k =$ `f` **then** $b :=$ **true**
   (c) **if** $b \wedge k \leq |r|$ **then** $h := h \vee (q \ll_{k-1} r)$

Let us now ask the question if $S0 \prec S2$. Then $q = [0, \texttt{f}, 0, i, \texttt{a}]$ and $r = [1]$. Since $|r| = 1$, line (2c) is never executed until $k = 1$. We reach $k = 1$ in the same state as in the previous example, but in this case $q \ll_0 r \equiv q_1 < r_1$ is true, and hence $S0 \prec S2$.

Although we have illustrated the algorithm with an example, the algorithm is not used in this fashion in the following sections. The important observation is that the algorithm only executes line (2c) at a subset of the dimensions. Moreover, the subset is determined purely structurally, i.e., given the AST and two statements, one can find a subset I where lexicographic comparison should be performed. This leads to the following re-formulation of the algorithm as an *incomplete* lexicographic order:

$$q \prec r \equiv \bigcup_{k \in I} q \ll_k r, \tag{8.11}$$

It is well known that the $\ll_p$ are disjoint. A pair q, r being given, there is at most one k such that $q \ll_k r$. k is the rightmost index such that $q[1 \ldots k] = r[1 \ldots k]$. From this follows that $\prec$ is transitive.

The observation that the "happens-before" relation is an incomplete lexicographic order becomes important in the next section. Because of this property, we can formulate the dataflow analysis questions for X10 programs in a way that can be efficiently solved.

Lastly, the way we have constructed Algorithm H clearly implies that:

Proposition 8.16 *Algorithm H exactly implements the "happens before" relation of Definition 8.8*

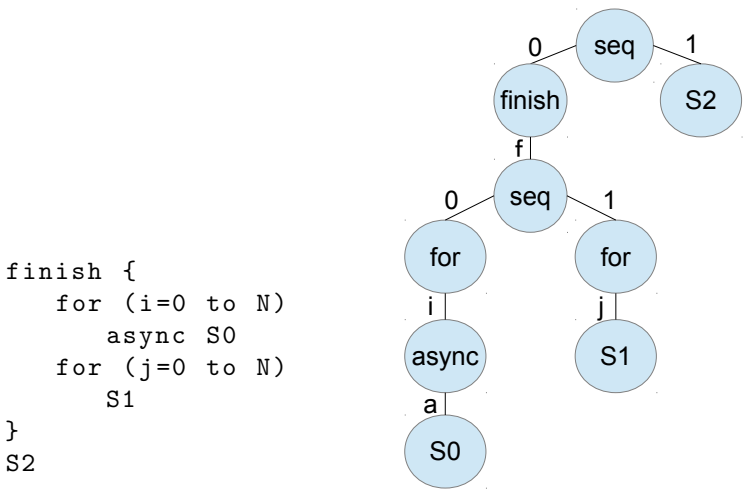


Figure 8.1: Example X10 code and its corresponding AST.

8.3 Dataflow Analysis

In this section, we present an adaptation of array dataflow analysis [30] for X10 programs, based on the "happens-before" as defined in the preceding section. The analysis is outlined in Figure 8.2.

Dataflow analysis aims at identifying, for each read access to a memory cell x, the *source* of the value found in x, i.e., the operation which wrote last into x. If the program is sequential, and fits into the polyhedral model [33] each read has a unique source which can be identified exactly [30].

However, the situation is different for parallel programs, since the actual execution order of operations may differ from run to run, due to scheduler decisions or hardware clock drifts. As a consequence, the content of some memory cell at a given step in the execution of a program may differ across runs. In other words, the answer to the dataflow question, which operation wrote last into x may be different.

This is called a race in software, or a hazard in hardware. The presence of a race condition in a program is usually a bug, which may be very difficult to diagnose and to correct, as it may manifest itself with very low probability.

8.3.1 Potential Sources

Let us focus on an instance of statement R at time stamp v, which has a read of array A at subscript(s) $f_R(v)$. The potential sources are instances of a statements W at time stamp w, which write into A at subscript(s) $f_W(w)$. f_R and f_W are vector functions of the same dimension as A.

Input:

R, A, f_R: A read in a statement R. R reads from a shared array (or scalar) A with access function f_R.

$\mathcal{W}$: Set of statements W that write to A with access function f_W.

Output:

Q: Quasi-Affine Solution Tree [29] that gives the producer of read by instances of R.

Algorithm:

1. **foreach** $W \in \mathcal{W}$
   (a) Compute Potential Sources $\Sigma_W(v)$ of W; the set of statement instances that (i) write to the same memory location, and (ii) the read do not happen before the write, parameterized by instances of R, v.
   (b) Compute Self Overwrites $\Sigma^-_W(v)$ of W; the set of writes overwritten by another instance of the same statement, parameterized by instances of R, v.
2. **foreach** $W \in \mathcal{W}$
   (a) Compute Validity Domain $Valid_W$ of $v \in R$; the set where v is valid for $\Sigma_W(v)$. An instance v becomes invalid if its write is overwritten by other statements.
3. Compute Q:
   $Q := \emptyset$
   foreach $W \in \mathcal{W}$
   (a) $Q := Q \cup \left((\Sigma_W(v) / \Sigma^-_W(v)) \wedge v \in Valid_W\right)$
   A write W is a producer of v if (i) it is a potential source, (ii) not overwritten by other instances of W, and (iii) not overwritten by writes of other statements.

Figure 8.2: Overview of array dataflow analysis for our X10 subset.

The set of potential sources is defined by:

$$v \in D_R, \tag{8.12}$$

$$w \in D_W, \tag{8.13}$$

$$f_W(w) = f_R(v), \tag{8.14}$$

$$\neg(v \prec w) \wedge v \neq w \tag{8.15}$$

Constraint (8.12) and (8.13) respectively constraints v and w to set of legal time stamps (iteration domain) of R and W. Constraint (8.14) restricts to those with conflicting memory accesses; those access the same *element* of the array A. Lastly, constraint (8.15) removes writes that happen after reads ($v \prec w$), and write by the same statement instance ($v = w$). In a sequential program, $\prec$ is total, hence $\neg(v \prec w) \wedge v \neq w \equiv w \prec v$, which is the usual formulation [30].

Let $\Sigma_W(v)$ be the set of potential sources as defined by (8.12-8.14) and (8.15). If the source program fits in the polyhedral model, all constraints are affine with the exception of (8.15), which can be expanded in Disjunctive Normal Form (DNF). Hence, $\Sigma_W(v)$ is a union of polyhedra.

8.3.2 Overwriting

Let x be a write to the same memory cell as w. x overwrite w if in all executions, x happens between w and v, or $w \prec x \prec v$. It is clear that if an overwrite exists, v will never see the value written by w. Both conditions are necessary: if one of them were not true, there would exists executions in which x happens before w, or v happens before x, and the value written at w would still be visible at v. This step is analogous to restricting the set of candidate sources to the most recent write in the original array dataflow analysis [30]. However, since the order is not total, the most recent write is not unique.

8.3.2.1 Self Overwrites

Write by a statement may be overwritten by other instances of the same statement. An instance w in $\Sigma_W(v)$ is a real source only if no other instance of W, x, overwrite w, i.e.,

$$w \in \Sigma_W(v) \wedge \neg\exists x \in \Sigma_W(v) : w \prec x. \tag{8.16}$$

This is exactly the definition of the set of upper bounds of Σ_W according to $\prec$. When $\prec$ is total, in the sequential case, (8.16) defines the unique maximum of P_W. In extreme cases, $\prec$ may be empty, and all tentative sources must be kept.

One possibility is to eliminate the existential quantifier in formula (8.16) using any projection algorithm, compute the negation and simplify the resulting formula. The drawback of this approach is its complexity: quantifier elimination in integers may generate expressions of exponential size, and so does negation.

Another possibility is to exploit the special form (8.11) of $\prec$. Since existential quantification distributes over disjunction, one has to compute $\exists x : w \ll_p x$ for each term which is present in $\prec$. Due to the very simple form of $\ll_p$, this set can be computed very efficiently using Parametric Integer Programming [29]. Simply solve the problem $\min\{x \in \Sigma_W(v) | w \ll_p x\}$ parametrically with respect to v and w. The result is a conditional expression (a *quast*) whose nodes bear affine constraints in the parameters, and whose leaves are either affine forms or the special term $\perp$, indicating that for some values of the parameters, the set above is empty. The disjunction of the paths leading to leaves not bearing a $\perp$ is the required projection. Then take the union of all such sets, denoted Σ_W^- in Figure 8.2, and subtract it from Σ_W.

8.3.2.2 Group Overwrites

Another case of overwrite happens when, for a given read, there are two possible writing statements, W_1 and W_2. A candidate source $w_1 \in \Sigma_{W_1}(v)$ is not visible to v if there exists $w_2 \in \Sigma_{W_2}(v)$ where $w_1 \prec w_2$.

This condition can be checked by inspecting the AST and Σ_W. In the AST, the paths from the root to W_1 and W_2 diverge at some `seq` or `async` or `finish` node. Assume that W_1 is to the left of W_2, then for each $w_1 \in \Sigma_{W_1}(v)$, one may associate w_2 such that $w_1 \prec w_2$. Let p be the common prefix of time stamps labeled to W_1 and W_2. Then $\forall k, 1 \leq k \leq p, w_1[k] = w_2[k]$ and $w_1[p+1] < w_2[p+1]$. Regardless of the remaining values beyond $p+1$, $w_1 \ll_p w_2$ holds. According to Algorithm H, $w_1 \prec w_2$ if the uppermost parallel construct in W_1 below the common node is not an `async`, and hence w_1 is overwritten by w_2. Otherwise, w_1 and w_2 can happen in any order and w_1 is not overwritten. Similarly, there is no w_2 where $w_1 \prec w_2$ if W_2 is to the left of W_1, since we obtain $w_2 \ll_p w_1$.

However, this construction fails if for the considered value of v, $\Sigma_{W_2}(v)$ is empty. As a set, $\Sigma_W(v)$ is a function of v, and it may be empty for some values of v, i.e., for some values of v, R may have no sources from W. Let us define the *range* of W as:

$$\Omega_W = \{v | \Sigma_W(v) \neq \emptyset\}.$$

The source is in $\Sigma_2(v)$ if $v \in \Omega_2$, and in $\Sigma_1(v)$ if $v \in \Omega_1 \backslash \Omega_2$.

In the general case we must consider more than two writing statements. Let us define an order on the writing statements by the definition: $W < W'$ if W is to the left of W' in the AST, and if the uppermost parallel construct in the path to W from the common node is not an `async`. It is easy to prove transitivity of $<$. For each W, one may define a validity by the rule:

$$\mathit{Valid}_W = \Omega_W \backslash \bigcup_{w<W'} \Omega_{W'}.$$

The source is in Σ_W only if $v \in \mathit{Valid}_W$. Maximal elements in the $<$ order have no successors, hence for them $\mathit{Valid}_W = \Omega_W$, while some other statements may have empty validity sets, indicating complete overwriting.

8.4 Race Detection

Once we have Σ_W, Σ_W^-, and $Valid_W$ for all writers W, the output quast of the dataflow analysis may be computed. However, the resulting quast may not be well-defined: a read may have multiple sources when the program contains races. The detection of races using the array dataflow analysis result is discussed in this section.

8.4.1 Race between Read and Write

The set of potential sources $\Sigma_W(v)$ can be split in two sets according to whether $w \prec v$ is true or not. If $\neg(w \prec v)$, $\neg(v \prec w)$ and $v \neq w$ (recall constraint (8.15)), then the read and the write may happen in parallel. In other words, v and w are not ordered, and thus v may execute before w in some execution but w may precede v in the other. Hence there is clearly a race in this case.

Let $\Sigma'_W(v)$ be like $\Sigma_W(v)$ with (8.15) replaced by $\neg(w \prec v) \wedge \neg(v \prec w) \wedge v \neq w$. A non empty $\Sigma'_W(v)$ indicates a race. The emptiness of Σ'_W may be tested in many way, for instance by expanding its constraints to DNF and applying linear programming, or by submitting its definition to an SMT solver, like Yices, Z3, or CVC among others. If Σ'_W is found to be non empty, the compiler may issue a warning about statement R, and no further analyses are needed.

8.4.2 Race between Writes

Let $\Sigma^*_W(v)$ the set of sources after self overwrites have been removed. Then the source of a read at v is $\Sigma^*_W(v)$ if $v \in Valid_W$. However, the source may not be unique if the program has races. There are two types of races:

- Race between multiple writes by the same statement. If there exists a solution to the problem $v \in Valid_W$, $x, y \in \Sigma^*_W(v)$, $x \neq y$, then x and y are involved in a race.
- Race between multiple writes by two statements. Two statements W_1 and W_2 have a race if: $Valid_{W_1} \wedge Valid_{W_2} \neq \emptyset$.

Both conditions can be checked by any SMT solver.

It is also important to note that it is not necessary to do all the above checks. For instance, if $W_1 < W_2$, their validity sets are disjoint by construction. Race detection of the first kind may be performed as we construct the sets, and the analysis may stop as soon as a race found. This approach may greatly reduce the complexity of the method.

8.4.3 Detection of Benign Races

The above approach is already sufficient to certify determinism of a program. However, additional analysis may be performed to flag questionable behavior of the program as warnings. For instance, our analysis detect array elements that are read but never written.

Another questionable behavior is *benign races*—races that do not influence the program determinacy. If two potential writes x and y may happen in parallel, x and y are in a race. However, if these writes are overwritten later or are not seen by any read, they are harmless. It might nevertheless be useful to warn the programmer of such behavior: a benign race can be taken as the indication of dead code. A way to handle them is therefore to do a backward recursive analysis starting from the output of the program. Any statement which is not encountered in this analysis is dead code and can be flagged as such.

8.4.4 Kernel Analysis

It is often the case that the full program is not polyhedral, while the core kernels are. In addition, due to the high cost of polyhedral analysis, it may not be practical to analyze the entire program.

The usual approach is to find "polyhedral parts"—subtrees in the AST that fit in the polyhedral model, and analyze them independently. Polyhedral methods are obvious candidates for such code fragments. For a finish/async language like X10, one must be more careful, since a subtree or a method may terminate but leave un-finished activities behind. Hence, to be handled with our methods, the sub-tree must satisfy the following properties in addition to the constraints of the polyhedral model:

- The uppermost parallel construct (in the path from the root of the sub-tree to each statement must be a `finish` if there is one.
- Similarly, let S be a statement that dominates statements in the sub-tree. Then the uppermost parallel construct in the path from the common prefix of S and the sub-tree to S must be a `finish` if there is one.

The above follows from Algorithm H, and ensures that all statements before and after the sub-tree are ordered by the happens-before relation, with respect to the statements in the sub-tree.

8.5 Examples

In this section, we illustrate by examples the importance of two key strength of our approach; statement instance-wise, and array element-wise analysis. We specifically compare with the work by Vechev et al. [120] and with other polyhedral approaches [23, 14]. We are not aware of any other state-of-the-art static analysis techniques for race detection that perform instance-wise or element-wise analysis.

8.5.1 Importance of Element-wise

Let us first use an example similar to the one used by Vechev et al. [120]. The following code is a simplified example of a common case in parallel programming, where a shared array is accesses by multiple threads.

```
finish {
  async for (i in 1..N)
      B[i] = C[i];          // S0
  async for (j in N..2*N)
      B[j] = C[2*i];        // S1
}
for (k=1:2N)
     ... = foo(B[k], ...); // S2
```

The time stamps and iteration domains are:

- $S0$: $[0, \texttt{f}, 0, \texttt{a}, i]$, $\mathcal{D}_{S0} = \{i | 1 \leq i \leq N\}$
- $S1$: $[0, \texttt{f}, 1, \texttt{a}, j]$, $\mathcal{D}_{S1} = \{j | N \leq j \leq 2N\}$
- $S2$: $[1, k]$, $\mathcal{D}_{S2} = \{k | 1 \leq k \leq 2N\}$

The only read in the program is the read of `B` by $S2$. Our analysis returns the following answer to the question: which statement produced the value of `B[k]` at $S2$:

- If $1 \leq k \leq N \wedge k \leq 2N$ then $S0[k]$ is a producer.
- If $1 \leq k \leq 2N \wedge N \leq k$ then $S1[k]$ is a producer.

where $Sn[v]$ denote the instance of Sn when its loop counters take the value v. It concludes that there is a race by two writers since the two sources overlap at $k = N$.

For this example, both of the other approaches will find the race with similar precision. However, if an analysis is not element-wise, then the analysis only finds that there is a race with the entire array `B`. Assuming that the programmer is warned of this race and change the lower bound of the j loop to $N + 1$, making the program race free, statement based approaches will still conservatively flag the array `B` to be in conflict.

8.5.2 Element-wise with Polyhedral

However, element-wise analysis in the work by Vechev et al. [120] is limited compared to polyhedral approaches, since they use an over-approximation. They require that any multi-dimensional arrays is reshaped into a 1D array, and the range of the 1D array to be represented with affine constraints. Furthermore, the renaming must be relative to what is called the `taskID` that identify an iteration of the loop ran by a thread.

For example, write to array `A` in the following code is expressed as writes to $\texttt{A}_\texttt{i}[\texttt{j}]$, where i is the `taskID`.

```
for (i in 0..(N-1))
  async
     for (j in 0..(N-1))
        A[i][j] = ... // S0
```

Approaches based on the polyhedral model, including ours, represents the write to `A` as an affine function $(i, j \rightarrow i, j)$ from the iteration domain $\mathcal{D}_{S0} = \{i, j | 0 \leq i, j < N\}$.

Let us illustrate the difference with a slight modification to the first example.

```
{ finish {
    async for (i in 0..(N-1))
        B[2*i] = C[i];       // S0
    async for (j in 0..(N-1))
        B[2*j+1] = C[2*i];  // S1
  }
  ... = foo(B[N]); // S2
}
```

The difference is in the writes to `B`, which now do not conflict. Our analysis returns the following answer to the question, which statement produced the value read by read `B[N]` at $S2$:

- If $\exists e : 2e = N \wedge N \geq 2$ then $S0[N/2]$ is a producer.
- If $\exists e : 2e = N - 1 \wedge N \geq 1$ then $S1[(N-1)/2]$ is a producer.

Note that the parametric integer linear programming [29] step (Section 8.3.2.1) introduces a "new parameter" (existentially quantified variable). The intersection of the two validity sets is empty, and we conclude that the program is race free.

However, the over-approximation by Vechev et al. [120] will approximate the write by $S0$ to be $0 \leq i \leq 2N-2$, and the write by $S1$ to be $1 \leq i \leq 2N-1$. Clearly, the two approximations overlap, and hence their approach would conservatively flag the program to have race.

8.5.3 Importance of Instance-wise

The examples above can also be implemented using *doall* loops. When implemented as parallel loops, previous approaches [23, 14] based on the polyhedral model can verify its determinacy, and does not require extensions proposed in this paper.

Our work can also detect races in finish/async programs that cannot be expressed with *doall* parallelism. The following is a simplified example of a case when such parallelism may be used. The example is based on Gauss-Seidel stencil computation that performs updates in-place, and uses some of the values (`A[i-1][j]` and `A[i][j-1]`) computed at the current time step and others from the previous time step.

The following code fragment illustrates a possible use of `async` in a way that cannot be expressed as loop parallelism. Detail of the statement `S1` is not given to simplify the presentation, but some code corresponding to an asynchronous send is the motivation behind this example.

```
for (t in 1..T)
   finish for (i in 1..N-2) {
      //boundary conditions omitted
      for (j in start..end)
        A[i][j] =
         update(A[i-1][j], A[i][j-1],
          A[i][j], A[i+1][j], A[i][j+1]); // S0
      async S1(A[i][end]);                // S1
      //boundary conditions omitted
   }
```

The point we illustrate with this example is the importance of statement instance-wise analysis. At the granularity of (static) statements, the pair of statements $S0, S1$ may happen in parallel. This is a conservative approximation because $S0[t, i]$ may happen in parallel with $S1[t', i']$ when $t \geq t'$ and $i > i'$. With this precision, our approach find that the read of `A[i][end]` by $S1$ is always the value written by $S0$.

8.5.4 Benefits of Array Dataflow Analysis

Array dataflow analysis is, strictly speaking, an overkill for detecting races. The formulation used by Vechev et al. [120] focuses on finding conflicting memory accesses. Dataflow analysis goes one step further by eliminating some of the accesses that are guaranteed to be overwritten from the consumer's perspective. Consider the following example:

```
finish{
  async{
    x = f(); //S1
    x = g(); //S2
  }
  async{
    x = h(); //S3
    x = k(); //S4
  }
}
t = x;       //S5
```

The approach by Vechev et al. [120] will find that statements $S1$, $S2$, $S3$, and $S4$ are all in race since they all may happen in parallel and writes to x. In contrast, array dataflow analysis will show that the read of x at $S5$ has two potential sources, $S2$ and $S4$.

The output by Vechev et al. [120] grows in size as the number of statements in `async` increases, while the output of dataflow analysis does not. When our analysis is integrated to a programming environment,

we believe that the preciseness and compactness of our analysis result will help the programmer more than simply detecting races.

Moreover, once the statement $S5$ is removed from the above example, the approach by Vechev et al. [120] would still detect a race, while our analysis would detect that the race is benign, and hence the full `finish` block is dead code.

8.6 Implementation

We have implemented our analysis for the subset of X10 described in Section 8.1. We take a representation of the AST, where statements only specify arrays (or scalars) being read or written, disregarding the what the operation is. Once we detect polyhedral regions in X10 programs, equivalent information can easily be extracted from the internal representation of the compiler.

Analysis of loop programs to detect regions amenable for polyhedral analysis, frequently referred to as Static Control Parts (SCoPs), or Affine Control Loops (ACLs) is well established through efforts to integrate polyhedral parallelizers into full compilers [39, 83]. In addition, we require that array accesses `a[i]` and `a[j]` point to the same memory location iff $i = j$. In general, such guarantee require pointer analysis, which is outside the scope of this paper.

We use the Integer Set Library [121] in our implementation to perform polyhedral operations and to solve parametric integer linear programming problems. The analysis itself is written in Java, and Java Native Interface is used to call ISL.

Java Grande Forum Benchmark Suite

Although our key contribution is verification of finish/async programs, we are not aware of any set of parallel benchmarks that use the extra expressive power of finish/async. We have demonstrated how our technique can handle such programs earlier with examples. In this section, we use Java Grande Forum benchmark suite [111] also used by Vechev et al. [120] to compare performance and applicability of our proposed analysis to their approach.

Out of the 8 benchmarks, 3 of them that were not handled by Vechev et al. [120] cannot be handled by ours either. Sparse includes indirect array accesses, which falls out of the polyhedral model. Similarly, MonteCarlo and RayTracer cannot be handle by polyhedral analysis. All of the remaining 5 at least partially fits the polyhedral model, and we were able to verify the determinacy of all parallel blocks. The result is summarized in Table 8.1.

Table 8.1: Performance of our implementation on JGF benchmarks [111]. Entries with the name followed by a number are verification of a parallel block that each contain a parallel loop surrounded by finish. All programs/blocks were verified to be determinate.

Benchmark	`while` **loop**[1]	**data-dep.** `if`[2]	**Time (s)**[4]	**Reference**[5] **Time (s)** [120]
CRYPT	Y		7.6	54.8
CRYPT1	Y		0.24	-
CRYPT2	Y		0.24	-
SOR			1.85	-
SOR1			0.29	0.41
LUFACT1			0.35	1.94
SERIES	Y		1.25	-
SERIES1	Y		0.06	55.8
MOLDYN1[3]			0.35	24.6
MOLDYN2		Y	0.92	2.5
MOLDYN3			0.14	0.32
MOLDYN4			0.08	1.01
MOLDYN5		Y	0.08	0.34

[1] Indicates that while loops were converted to for loops. These while loops are of the form:
`n=100; do { ... n--;} while (n>=0);`

[2] Indicates that data-dependent if statements were over-approximated by assuming both branches were always taken.

[3] MOLDYN require a final variable *pad* to be constant propagated due to expressions like: $i2 * pad$.

[4] Our experiments were conducted with 4-core Intel Core2Quad (2.83GHz) and 8GB of memory. We used Java 1.6, and ISL 0.10.

[5] These timing results are taken from the article by Vechev et al. [120] and were conducted with 4-core Xeon (3.8GHz) and 5GB of memory. However, their implementation is not directly comparable to ours, due to many reasons. For example, they work on a lower level representation of the program (`Jimple`).

8.7 Related Work

Array Dataflow Analysis [30, 87] is the key analysis that connects loop programs and polyhedral representaions. Array dataflow analysis was introduced by Feautrier [30] and further expanded by Pugh and Wonnacott [88]. Extensions beyond the polyhedral model were proposed by Pugh and Wonnacott [89] and by Barthou et al. [12]. As far as we know, multi-dimensional time stamps were first introduced by Feautrier [32] as a trick for proving the existence of schedules for well-structured sequential programs. They were further exploited for specifying complex program transformations by Bastoul [13]. They are similar to the *pedigrees* proposed by Leiserson et al. [70], with the difference that pedigrees are computed at run time, while time stamps exist only at compile time.

Since the emphasis in the polyhedral literature is placed on automatic parallelization, there has been very little work on verifying already parallel programs. The work by Collard and Griebl [23] that presents array dataflow analysis for programs with *doall* parallelism is most closely related to our work. The key distinction is that we handle parallel constructs in the X10 language, `finish` and `async` that can express parallelism not expressible by doall loops.

The result of array dataflow analysis is the answer to the question, which instance of which statement produced the value used. For sequential programs, the result should be a unique statement instance (except for input dependences). However, when the input program is parallel, multiple statement instances may happen in parallel, and hence the result may not be unique. Thus, we may take the result of array dataflow analysis for X10 programs, and detect races by checking if the producers are uniquely identified.

One key question in reasoning about determinism is the question: which statements (or statement instances) have a clearly defined order of execution. Analyses to answer this question for finish/async programs, closely related to our "happens-before" relation, have been presented by Agarwal et al. [5] and by Lee and Palsberg [66].

While Lee and Palsberg work at the level of a statement, Agarwal et al. try to increase precision by exhibiting conditions on loop counters that guarantee (or forbid) parallel execution. However, these conditions use only equality and inequality, instead of the full power of affine constraints. The algorithms in these two papers are surprisingly complex when compared to our work.

There is a separate body of work that address race detection with multi-threaded programs with locks (e.g., [28, 49]). These methods are not directly applicable to modern parallel languages where locks are rarely used.

The work by Vechev et al. [120] goes beyond the evaluation of the "may happens in parallel" relation and attempts to verify determinism of finish/async programs. Their analysis is also instance-wise and element-wise. The main difference is that their work uses over-approximations of memory accesses, where our analysis

is exact. In addition, we use array dataflow analysis to find races, but the information given by the analysis, which is more than enough to find races, can be used for other purposes.

Dynamic race detection (e.g., [108]) is a complementary technique to static analysis, and is more broadly applicable. However, dynamic analysis requires significant run-time overhead, and is subject to the well-known Dijkstra saying, that they can be used to prove the existence of races, not their absence.

8.8 Discussion

The contributions presented in this chapter are in a very different direction compared to those presented in earlier chapters. This is a first step towards applying polyhedral analysis to finish/async programs. It has been written in the context of the X10 language. However, we expect our approach to be applicable to other languages with similar parallel constructs. For programs that fit in the polyhedral model, the analysis is exact, and as precise as can be. There may be neither false positives nor false negatives.

Although we have focused on race detection, the adaptation of dataflow analysis have much more potential applications. These potential applications include scheduling and locality improvement, undefined variables detection, constant propagation and semantic program verification.

The approach presented in this chapter can be extended in two directions:

- The X10 language has several control constructs which may create (or remove) races. Among them are clocks, a generalization of the classical barriers, the `atomic` modifier, and the `at` statement, which delegates a calculation to a remote place of the target system. Basically, all these constructs necessitate a new definition of the "happens before" relation. The question is whether Algorithm H can be extended to take care of them.

  Handling `atomic` and `at` constructs is a minor extension to the results presented here, but space constraints preclude an elaborate explanation. We are currently working on extending our analysis to handle clocks.

- Like all polyhedral analyses, our method applies only to a limited class of programs. Is there a possibility to remove some of these restrictions? A classical approach is to deal only with polyhedral subtrees of the AST, provided they don't interfere with the remnants of the program. Since X10 is an object oriented language, it might be possible to find many self-contained polyhedral methods.

  One may also resort to approximations. The difficulty here is that since the source computation uses set *differences* (see for instance Section 8.3.2.2) over- and under-approximations are both needed. Depending on the quality of the approximations, the resulting analysis may have both false negatives and false positives. The problem will be to minimize their number.

Chapter 9

Conclusions

Driven by power and heat problems, further increase in compute power now relies on parallelism, and this trend is not expected to change in the near future. Ideally, programmers can write simple programs that merely specify what needs to be computed, and let the compiler optimize for performance. However, programmers still strive to improve performance by writing complicated codes, even for sequential programs.

The polyhedral model is a very powerful framework for program transformations, and plays a central role in all of our contributions. Even after many years of research, only a small subset of possible transformations via the model has been explored. We have shown various applications of the model: memory re-allocation, complexity reduction, distributed memory parallelization, and determinacy guarantees. The rich analysis capability of the polyhedral formalism will continue to serve as a fertile ground to develop powerful transformations.

There are many direct extensions to our work. The design space explored by AlphaZ is still a small subset, and can be expanded in various directions. We will need new code generators to explore efficient execution strategies in new architectures, such as GPGPUs and Many Integrated Cores. We are currently working on extending the Target Mapping to include other important optimizations, such as multi-level tiling and vectorization. The embedding we describe in Chapter 5 is largely based on heuristics, and can be improved. When there is a specific goal, i.e., to have one dimension that is uniform, how to perform embedding and uniformization to satisfy such goal, as opposed to trying to uniformize everything, remains an open problem. The MPI parallelization can be combined with OpenMP by applying additional levels of tiling.

We have restricted ourselves to partially uniform programs for our distributed memory code generation. Affine dependences are much more expressive, but dependences in polyhedral programs are often simple expressions. For instance, one rarely sees dependences with coefficients on indices greater than 1. We feel that it is important to look for other classes of dependences between affine and uniform, that are more expressive than uniform, but easier to analyze than affine. As we have shown in Chapter 7, assuming less general dependences can significantly simplify the problem, and it may not be a huge restriction in practice. One candidate for such class of dependence is that described by Lamport [61], where the linear part of a dependence is an arbitrary permutation of some subset of the indices.

Our work on polyhedral X10 programs opens up a number of interesting future directions. It is an extension to the polyhedral model to handle more than *doall* type loop parallelism. Automatic parallelization

in the polyhedral model has always used *doall* parallelism in the past, and now a number of questions: how to schedule, how to generate code, how tiling is affected, and so on, must be re-visited for finish/async programs. In addition, we have not yet handled another important parallel construct in X10; `clocks`, which we are currently working on.

The main drawback of polyhedral techniques is its limited applicability. Although affine control programs are an important class of programs, there are many others that do not fit the model. Our extension to array dataflow analysis for X10 programs is an attempt to enlarge the applicability. However, we only expand the expressiveness of parallelism in the polyhedral model, and we still require controls to be affine.

There is much more to be explored in the current scope of the polyhedral model. At the same time, it is important to look for extensions to the polyhedral model to larger class of programs. It is extremely difficult to expand the model to larger class of programs, and also retain the same precision. However, with some form of approximations, the model can be applied in many more programs. For example, we can reduce the granularity and reason about blocks of computations rather than statements, and only require inter-block dependences to be affine. We may not be able to perform some analysis statically, but can still combine compile-time knowledge to improve run-time optimizations. These ideas are not necessarily new, but have not been seriously pursued in the past.

The possible applications of the core techniques underlining the polyhedral model; linear algebra based formalism, parametric integer linear programming, and all other polyhedral ways of looking at a program; are not limited to program transformations. As an example, it has been previously applied to analyze program termination [6]. We believe that there are more applications of the polyhedral model outside its traditional use in parallelization.

References

[1] Eclipse modeling framework. `http://www.eclipse.org/modeling/emf/`.

[2] Tom/gom. `http://tom.loria.fr/`.

[3] Xtend. `http://www.eclipse.org/Xtend/`.

[4] Xtext. `http://www.eclipse.org/Xtext/`.

[5] S. Agarwal, R. Barik, V. Sarkar, and R. K. Shyamasundar. May-happen-in-parallel analysis of X10 programs. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '07, pages 183–193, 2007.

[6] C. Alias, A. Darte, P. Feautrier, and L. Gonnord. Multi-dimensional rankings, program termination, and complexity bounds of flowchart programs. In *Proceedings of the 17th International Conference on Static Analysis*, SAS '10, pages 117–133, 2010.

[7] E. Allen, D. Chase, J. Hallett, V. Luchangco, J. Maessen, S. Ryu, G. Steele Jr, S. Tobin-Hochstadt, J. Dias, C. Eastlund, et al. The Fortress Language Specification. *Sun Microsystems*, 139:140, 2005.

[8] S. P. Amarasinghe and M. S. Lam. Communication optimization and code generation for distributed memory machines. In *Proceedings of the 14th ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI '93, pages 126–138, 1993.

[9] R. Andonov, S. Balev, S. Rajopadhye, and N. Yanev. Optimal semi-oblique tiling. In *Proceedings of the 13th ACM Symposium on Parallel Algorithms and Architectures*, SPAA '01, pages 153–162, 2001.

[10] F. André, M. Fur, Y. Mahéo, and J.-L. Pazat. The pandore data-parallel compiler and its portable runtime. In B. Hertzberger and G. Serazzi, editors, *High-Performance Computing and Networking*, volume 919 of *Lecture Notes in Computer Science*, pages 176–183. 1995.

[11] P. Banerjee, J. Chandy, M. Gupta, E. Hodges IV, J. Holm, A. Lain, D. Palermo, S. Ramaswamy, and E. Su. The Paradigm compiler for distributed-memory multicomputers. *Computer*, 28(10):37–47, 1995.

[12] D. Barthou, J.-F. Collard, and P. Feautrier. Fuzzy array dataflow analysis. *Journal of Parallel and Distributed Computing*, 40:210–226, 1997.

[13] C. Bastoul. Code generation in the polyhedral model is easier than you think. In *Proceedings of the 13th IEEE International Conference on Parallel Architecture and Compilation Techniques*, PACT '04, pages 7–16, 2004.

[14] V. Basupalli, T. Yuki, S. Rajopadhye, A. Morvan, S. Derrien, P. Quinton, and D. Wonnacott. ompverify: polyhedral analysis for the OpenMP programmer. *OpenMP in the Petascale Era*, pages 37–53, 2011.

[15] U. Bondhugula. Automatic distributed-memory parallelization and code generation using the polyhedral framework. Technical report, IISc Research Report, IISc-CSA-TR-2011-3, 2011.

[16] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 101–113, 2008.

[17] B. Chamberlain, D. Callahan, and H. Zima. Parallel programmability and the Chapel language. *International Journal of High Performance Computing Applications*, 21(3):291–312, 2007.

[18] A. Chandrakasan, S. Sheng, and R. Brodersen. Low-power cmos digital design. *IEICE Transactions on Electronics*, 75(4):371–382, 1992.

[19] C. Chen, J. Chame, and M. Hall. Chill: A framework for composing high-level loop transformations. Technical report, University of Southern California, 2008.

[20] Z. Chen and W. Shang. On uniformization of affine dependence algorithms. In *Proceedings of the 4th IEEE Symposium on Parallel and Distributed Processing*, pages 128–137, 1992.

[21] M. Claßen and M. Griebl. Automatic code generation for distributed memory architectures in the polytope model. In *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium*, IPDPS '06, page 243, 2006.

[22] S. Coleman and K. McKinley. Tile size selection using cache organization and data layout. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming language Design and Implementation*, PLDI '95, pages 279–290, 1995.

[23] J.-F. Collard and M. Griebl. Array dataflow analysis for explicitly parallel programs. In L. Bougé, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *Euro-Par'96 Parallel Processing*, volume 1123 of *Lecture Notes in Computer Science*, pages 406–413. 1996.

[24] D. Culler, R. Karp, D. Patterson, A. Sahay, K. Schauser, E. Santos, R. Subramonian, and T. Von Eicken. *LogP: Towards a realistic model of parallel computation*, volume 28. 1993.

[25] A. Darte, R. Schreiber, and G. Villard. Lattice-based memory allocation. *IEEE Transactions on Computers*, 54(10):1242–1257, 2005.

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

[27] F. Desprez. *Procdures de base pour le calcul scientifique sur machines parallles mmoire distribue*. PhD thesis, Institut National Polytechnique de Grenoble, 1994. LIP ENS-Lyon.

[28] D. Engler and K. Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. *ACM SIGOPS Operating Systems Review*, 37(5):237–252, 2003.

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

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

[31] P. Feautrier. Some efficient solutions to the affine scheduling problem, I, one-dimensional time. *International Journal of Parallel Programming*, 21(5):313–348, 1992.

[32] P. Feautrier. Some efficient solutions to the affine scheduling problem, II, multidimensional time. *International Journal of Parallel Programming*, 21(6):389–420, 1992.

[33] P. Feautrier and C. Lengauer. The polyhedral model. In D. Padua, editor, *Encyclopedia of Parallel Programming*. 2011.

[34] A. Floch, T. Yuki, C. Guy, S. Derrien, B. Combemale, S. Rajopadhye, and R. France. Model-driven engineering and optimizing compilers: A bridge too far? In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems*, MODELS '11, 2011.

[35] R. France and B. Rumpe. Model-driven development of complex software: A research roadmap. In *Proceedings of the 2007 Future of Software Engineering*, FOSE '07, pages 37–54, 2007.

[36] M. L. Fur, J.-L. Pazat, and F. André. An array partitioning analysis for parallel loop distribution. In *Proceedings of the First International Euro-Par Conference on Parallel Processing*, Euro-Par '95, pages 351–364, 1995.

[37] G. Goumas, N. Drosinos, M. Athanasaki, and N. Koziris. Message-passing code generation for non-rectangular tiling transformations. *Parallel Computing*, 32(10):711–732, 2006.

[38] M. Griebl, P. Feautrier, and C. Lengauer. Index set splitting. *International Journal of Parallel Programming*, 28(6):607–631, 2000.

[39] T. Grosser, H. Zheng, R. Aloor, A. Simbürger, A. Größlinger, and L. Pouchet. Polly–Polyhedral optimization in LLVM. In *1st International Workshop on Polyhedral Compilation Techniques*, IMPACT '11, 2011.

[40] G. Gupta and S. Rajopadhye. Simplifying reductions. In *Proceedings of the 33rd ACM Conference on Principles of Programming Languages*, PoPL '06, pages 30–41, 2006.

[41] G. Gupta, S. Rajopadhye, and P. Quinton. Scheduling reductions on realistic machines. In *Proceedings of the 14th Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '02, pages 117–126, 2002.

[42] M. Gupta and P. Banerjee. Demonstration of automatic data partitioning techniques for parallelizing compilers on multicomputers. *IEEE Transactions on Parallel and Distributed Systems*, 3:179–193, 1992.

[43] A. Hartono, M. M. Baskaran, C. Bastoul, A. Cohen, S. Krishnamoorthy, B. Norris, J. Ramanujam, and P. Sadayappan. Parametric multi-level tiling of imperfectly nested loops. In *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, pages 147–157, 2009.

[44] A. Hartono, M. M. Baskaran, J. Ramanujam, and P. Sadayappan. Dyntile: Parametric tiled loop generation for parallel execution on multicore processors. In *Proceedings of the 24th IEEE International Symposium on Parallel and Distributed Processing*, pages 1–12, 2010.

[45] High Performance Fortran Forum. High performance fortran language specification. 1993.

[46] F. Irigoin, P. Jouvelot, and R. Triolet. Semantical interprocedural parallelization: An overview of the pips project. In *Proceedings of the 5th International Conference on Supercomputing*, ICS '91, pages 244–251, 1991.

[47] F. Irigoin and R. Triolet. Supernode partitioning. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, PoPL '88, pages 319–329, 1988.

[48] IRISA, CAIRN. Generic Compiler Suite. `http://gecos.gforge.inria.fr/`.

[49] V. Kahlon, N. Sinha, E. Kruus, and Y. Zhang. Static data race detection for concurrent programs with asynchronous calls. In *Proceedings of the the 7th joint meeting of the European software engineering Conference and the ACM SIGSOFT Symposium on The foundations of software engineering*, pages 13–22, 2009.

[50] R. Karp, R. Miller, and S. Winograd. The organization of computations for uniform recurrence equations. *Journal of the ACM*, 14(3):563–590, 1967.

[51] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott. The omega calculator and library, version 1.1. 0. Technical report, Department of Computer Science, University of Maryland, College Park, 1996.

[52] W. Kelly, W. Pugh, and E. Rosser. Code generation for multiple mappings. In *Proceedings of the Fifth Symposium on the Frontiers of Massively Parallel Computation*, pages 332–341, 1995.

[53] D. Kim. *Parameterized and Multi-level Tiled Loop Generation.* PhD thesis, Colorado State University, Fort Collins, CO, USA, 2010.

[54] D. Kim and S. Rajopadhye. Efficient tiled loop generation: D-tiling. In *The 22nd International Workshop on Languages and Compilers for Parallel Computing*, LCPC '09, 2009.

[55] D. Kim, L. Renganarayanan, D. Rostron, S. Rajopadhye, and M. Strout. Multi-level tiling: M for the price of one. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, SC '07, page 51, 2007.

[56] N. Kim, T. Austin, D. Baauw, T. Mudge, K. Flautner, J. Hu, M. Irwin, M. Kandemir, and V. Narayanan. Leakage current: Moore's law meets static power. *Computer*, 36(12):68–75, 2003.

[57] D. Kranz, K. Johnson, A. Agarwal, J. Kubiatowicz, and B. Lim. *Integrating message-passing and shared-memory: early experience*, volume 28. 1993.

[58] S. Krishnamoorthy, M. Baskaran, U. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan. Effective automatic parallelization of stencil computations. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 235–244, 2007.

[59] O. Kwon, F. Jubair, S. Min, H. Bae, R. Eigenmann, and S. Midkiff. Automatic scaling of openmp beyond shared memory. In *Proceedings of the 24th International Workshop on Languages and Compilers for Parallel Computing*, LCPC '11, 2011.

[60] M. Lam, E. Rothberg, and M. Wolf. The cache performance and optimizations of blocked algorithms. volume 25 of *PLDI '91*, pages 63–74, 1991.

[61] L. Lamport. The parallel execution of DO loops. *Communications of the ACM*, 17(2):83–93, 1974.

[62] H. Le Verge. Reduction operators in alpha. In D. Etiemble and J.-C. Syre, editors, *Parallel Algorithms and Architectures, Europe*, pages 397–411, Paris, 1992. See also, Le Verge Thesis (in French).

[63] H. Le Verge. Recurrences on lattice polyhedra and their applications to the synthesis of systolic arrays. This was the last document that Herve Le Verge worked on before his untimely death. An updated version will appear as an IRISA research report, Jan 1994.

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

[65] H. Le Verge and P. Quinton. Un environnement de transformations de programmes pour la synthèse d'architectures régulières. 1992.

[66] J. K. Lee and J. Palsberg. Featherweight X10: a core calculus for async-finish parallelism. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 25–36, 2010.

[67] P. Lee. Efficient algorithms for data distribution on distributed memory parallel computers. *IEEE Transactions on Parallel and Distributed Systems*, 8:825–839, 1997.

[68] P. Lee and Z. M. Kedem. Automatic data and computation decomposition on distributed memory parallel computers. *ACM Transaction on Programming Languages and Systems*, 24:20, 1999.

[69] V. Lefebvre and P. Feautrier. Automatic storage management for parallel programs. *Parallel Computing*, 24(3-4):649–671, 1998.

[70] C. E. Leiserson, T. B. Schardl, and J. Sukha. Deterministic parallel random-number generation for dynamic-multithreading platforms. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 193–204, 2012.

[71] J. Li and M. Chen. Generating explicit communication from shared-memory program references. In *Proceedings of the 1990 ACM/IEEE Conference on Supercomputing*, SC '90, pages 865–876, 1990.

[72] J. Li and M. Chen. Compiling communication-efficient programs for massively parallel machines. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):361–376, 1991.

[73] R. Lyngsø, M. Zuker, C. Pedersen, et al. Fast evaluation of internal loops in rna secondary structure prediction. *Bioinformatics*, 15(6):440–445, 1999.

[74] M. Manjunathaiah, G. Megson, S. Rajopadhye, and T. Risset. Uniformization of affine dependence programs for parallel embedded system design. In *Proceedings of the 30th International Conference on Parallel Processing*, ICPP '01, pages 205–213, 2001.

[75] N. Markham and M. Zuker. Software for nucleic acid folding and hybridization. *Methods in Molecular Biology*, 453:3–31, 2008.

[76] C. 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.

[77] D. Maydan, S. Amarasinghe, and M. Lam. Array-data flow analysis and its use in array privatization. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, PoPL '93, pages 2–15, 1993.

[78] B. Meister, A. Leung, N. Vasilache, D. Wohlford, C. Bastoul, and R. Lethin. Productivity via automatic code generation for PGAS platforms with the R-Stream compiler. In *Proceedings of the Workshop on Asynchrony in the PGAS Programming Model*, 2009.

[79] J. Mellor-Crummey, V. Adve, B. Broom, D. Chavarrıa-Miranda, R. Fowler, G. Jin, K. Kennedy, and Q. Yi. Advanced optimization strategies in the rice dhpf compiler. *Concurrency and Computation: Practice and Experience*, 14(8-9):741–767, 2002.

[80] R. W. Numrich and J. Reid. Co-array fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, 1998.

[81] N. Osheim, M. Strout, D. Rostron, and S. Rajopadhye. Smashing: Folding space to tile through time. In *Proceedings of the 21th International Workshop on Languages and Compilers for Parallel Computing*, LCPC '07, pages 80–93, 2008.

[82] T. Pathan. RNA secondary structure prediction using AlphaZ. Master's thesis, Colorado State University, Fort Collins, CO, USA, 2010.

[83] S. Pop, A. Cohen, C. Bastoul, S. Girbal, G. Silber, and N. Vasilache. GRAPHITE: Polyhedral analyses and optimizations for GCC. In *Proceedings of the 2006 GCC Developers Summit*, 2006.

[84] L.-N. Pouchet. PolyBench. `www.cs.ucla.edu/~pouchet/software/polybench/`.

[85] L.-N. Pouchet, C. Bastoul, A. Cohen, and N. Vasilache. Iterative optimization in the polyhedral model: Part I, one-dimensional time. In *IEEE/ACM Fifth International Symposium on Code Generation and Optimization*, CGO '08, pages 144–156, 2007.

[86] L.-N. Pouchet, U. Bondhugula, C. Bastoul, A. Cohen, J. Ramanujam, and P. Sadayappan. Hybrid iterative and model-driven optimization in the polyhedral model. Technical Report 6962, INRIA Research Report, 2009.

[87] W. Pugh. A practical algorithm for exact array dependence analysis. *Communications of the ACM*, 35(8):102–114, 1992.

[88] W. Pugh and D. Wonnacott. Eliminating false data dependences using the Omega test. In *Proceedings of the 13th ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI '92, pages 140–151, 1992.

[89] W. Pugh and D. Wonnacott. Going beyond Integer Programming with the Omega Test to Eliminate False Data Dependencies. Technical Report CS-TR-3191, University of Maryland, 1992.

[90] F. Quilleré and S. Rajopadhye. Optimizing memory usage in the polyhedral model. *ACM Transactions on Programming Languages and Systems*, 22(5):773–815, 2000.

[91] F. Quilleré, S. Rajopadhye, and D. Wilde. Generation of efficient nested loops from polyhedra. *International Journal of Parallel Programming*, 28(5):469–498, 2000.

[92] M. Quinn and P. Hatcher. On the utility of communication-computation overlap in data-parallel programs. *Journal of Parallel and Distributed Computing*, 33(2):197–204, 1996.

[93] M. Rahman, L. Pouchet, and P. Sadayappan. Neural network assisted tile size selection. In *Proceedings of the International Workshop on Automatic Performance Tuning*, 2010.

[94] S. Rajopadhye, G. Gupta, and D. Kim. Alphabets: An Extended Polyhedral Equational Language. In F. Nakano, Bordim, editor, *Proceedings of the 13th Workshop on Advances in Parallel and Distributed Computational Models*, 2011.

[95] S. V. Rajopadhye, S. Purushothaman, and R. M. Fujimoto. On synthesizing systolic arrays from recurrence equations with linear dependencies. In *Proceedings of the 6th Conference on Foundations of software technology and theoretical computer science*, pages 488–503, 1986.

[96] J. Ramanujam and P. Sadayappan. Tiling of iteration spaces for multicomputers. In *Proceedings of the 1990 International Conference on Parallel Processing*, volume 2 of *ICPP '90*, pages 179–186, 1990.

[97] M. Ravishankar, J. Eisenlohr, L. Pouchet, J. Ramanujam, A. Rountev, and P. Sadayappan. Automatic parallelization of a class of extended affine loop nests for distributed memory systems. Technical report, OSU-CISRC-5/12-TR10, Ohio State University, 2012.

[98] L. Renganarayana, U. Bondhugula, S. Derisavi, A. Eichenberger, and K. O'Brien. Compact multi-dimensional kernel extraction for register tiling. In *Proceedings of the 2009 ACM/IEEE Conference on Supercomputing*, SC '09, page 45, 2009.

[99] L. Renganarayana and S. Rajopadhye. Positivity, posynomials and tile size selection. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 1–12, 2008.

[100] L. Renganarayanan. *Scalable and Efficient Tools for Multi-level Tiling*. PhD thesis, Colorado State University, Fort Collins, CO, USA, 2008.

[101] L. Renganarayanan, D. Kim, S. Rajopadhye, and M. M. Strout. Parameterized tiled loops for free. In *Proceedings of the 28th ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI '07, pages 405–414, 2007.

[102] G. Rivera and C. wen Tseng. A comparison of compiler tiling algorithms. In *Proceedings of the 8th International Conference on Compiler Construction*, pages 168–182, 1999.

[103] J. Roelofs and M. Strout. Tiling Visualizer, 2008. `http://www.cs.colostate.edu/~mstrout/`.

[104] V. P. Roychowdhury. *Derivation, extensions and parallel implementation of regular iterative algorithms*. PhD thesis, Stanford University, Stanford, CA, USA, 1989.

[105] V. Saraswat, B. Bloom, I. Peshansky, O. Tardieu, and D. Grove. X10 language specification version 2.2, Mar. 2012. `x10.sourceforge.net/documentation/languagespec/x10-latest.pdf`.

[106] V. Saraswat and R. Jagadeesan. Concurrent clustered programming. In *CONCUR 2005 - Concurrency Theory*, pages 353–367, 2005.

[107] V. A. Saraswat, R. Jagadeesan, M. Michael, and C. von Praun. A theory of memory models. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '07, pages 161–172, 2007.

[108] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[109] R. Schreiber and J. J. Dongarra. Automatic blocking of nested loops. Technical report, Oak Ridge National Laboratory, 1990.

[110] T. Shen and D. Wonnacott. Code generation for memory mappings. In *Proceedings of the 1998 Mid-Atlantic Student Workshop on Programming Languages and Systems*, 1998.

[111] L. Smith, J. Bull, and J. Obdrizalek. A parallel Java Grande benchmark suite. In *Proceedings of the ACM/IEEE 2001 Conference on Supercomputing*, SC '01, pages 6–6, 2001.

[112] M. Strout, L. Carter, J. Ferrante, and B. Simon. Schedule-independent storage mapping for loops. *ACM SIGOPS Operating Systems Review*, 32(5):24–33, 1998.

[113] A. Sussman. Model-driven mapping onto distributed memory parallel computers. In *Proceedings of the 1992 ACM/IEEE Conference on Supercomputing*, SC '92, pages 818–829, 1992.

[114] S. Tavarageri, L. Pouchet, J. Ramanujam, A. Rountev, and P. Sadayappan. Dynamic selection of tile sizes. In *Proceedings of the 18th International Conference on High Performance Computing*, pages 1–10, 2011.

[115] W. Thies, F. Vivien, J. Sheldon, and S. Amarasinghe. A unified framework for schedule and storage optimization. In *Proceedings of the 22nd International Conference on Programming Language Design and Implementation*, PLDI '01, pages 232–242, 2001.

[116] UPC Consortium et al. UPC language specifications. *Lawrence Berkeley National Lab Tech Report LBNL–59208*, 2005.

[117] V. Van Dongen and P. Quinton. Uniformization of linear recurrence equations: a step toward the automatic synthesis of systolic arrays. In *Proceedings of the International Conference on Systolic Arrays*, pages 473–482, 1988.

[118] N. Vasilache. *Scalable Program Optimization Techniques In The Polyhedral Model*. PhD thesis, University of Paris-Sud 11, 2007.

[119] N. Vasilache, B. Meister, A. Hartono, M. Baskaran, D. Wohlford, and R. Lethin. Trading off memory for parallelism quality. In *2nd International Workshop on Polyhedral Compilation Techniques*, IMPACT '12, 2012.

[120] M. Vechev, E. Yahav, R. Raman, and V. Sarkar. Automatic verification of determinism for structured parallel programs. In *Proceedings of the 17th International Conference on Static Analysis*, SAS '10, pages 455–471, 2010.

[121] S. Verdoolaege. isl: An integer set library for the polyhedral model. *Mathematical Software–ICMS 2010*, pages 299–302, 2010.

[122] D. Wilde and S. Rajopadhye. The naive execution of affine recurrence equations. In *Proceedings of the 1995 International Conference on Application Specific Array Processors*, ASAP '95, pages 1–12, 1995.

[123] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *Proceedings of the 12th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '91, pages 30–44, 1991.

[124] M. Wolfe. Iteration space tiling for memory hierarchies. In *Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing*, pages 357–361, 1987.

[125] D. Wonnacott. Achieving scalable locality with time skewing. *International Journal of Parallel Programming*, 30(3):181–221, 2002.

[126] P. Wu, P. Feautrier, D. Padua, and Z. Sura. Instance-wise points-to analysis for loop-based dependence testing. In *Proceedings of the 16th International Conference on Supercomputing*, pages 262 – 273, 2002.

[127] W. Wulf and S. McKee. Hitting the memory wall: Implications of the obvious. *ACM SIGARCH Computer Architecture News*, 23(1):20–24, 1995.

[128] Y. Yaacoby and P. Cappello. Converting affine recurrence equations to quasi-uniform recurrence equations. In J. Reif, editor, *VLSI Algorithms and Architectures*, volume 319 of *Lecture Notes in Computer Science*, pages 319–328. 1988.

[129] Y. Yaacoby and P. Cappello. Converting affine recurrence equations to quasi-uniform recurrence equations. In J. Reif, editor, *VLSI Algorithms and Architectures*, volume 319 of *Lecture Notes in Computer Science*, pages 319–328. 1988.

[130] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, et al. Titanium: A high-performance Java dialect. *Concurrency Practice and Experience*, 10(11-13):825–836, 1998.

[131] Q. Yi. Poet: a scripting language for applying parameterized source-to-source program transformations. *Software: Practice and Experience*, 2011.

[132] T. Yuki, L. Renganarayanan, S. Rajopadhye, C. Anderson, A. Eichenberger, and K. O'Brien. Automatic creation of tile size selection models. In *Proceedings of the 8th IEEE ACM International Symposium on Code Generation and Optimization*, CGO '10, pages 190–199, 2010.