THESIS

THE ALPHAZ VERIFIER

Submitted by

Vamshi Basupalli

Department of Computer Science

In partial fulfillment of the requirements

for the Degree of Master of Science

Colorado State University

Fort Collins, Colorado

Fall 2011

Master's Committee:

Advisor : Sanjay Rajopadhye

Michelle Strout
Sudeep Pasricha

ABSTRACT

THE ALPHAZ VERIFIER

In the context of a compiler research framework, it is useful to have a tool that can certify that a proposed transformation is legal according to the semantics of the source language and a model of the target architecture (sequential, SIMD, shared memory, etc.) This thesis describes such a tool, the AlphaZ Verifier developed for a system for parallelizing and transforming programs in Alphabets, a high level polyhedral equational language. Our Verifier takes an alphabets program and a proposed target mapping as input. The mapping is very general and includes a proposed space-time mapping, a memory mapping, and a tiling specification (specifically which of the dimensions after the space-time mapping are to be tiled). The Verifier first checks whether the space-time mapping is legal, i.e., no dependences are violated. Next it validates the memory mapping by ensuring that no live values are overwritten before their final use. Finally, it checks tiling legality by ensuring that the proposed tiling dimensions are fully permutable. We show how all the necessary checks are reduced to polyhedron emptiness checking so that a single back-end engine can be used for each of these steps.

TABLE OF CONTENTS

Chapter 1

Introduction

With the arrival of multi-cores and many-cores into the mainstream, high performance computing is gaining importance like never before. Having said that, parallel programming and optimization is still considered a hard task and done mostly by high performance computing (HPC) specialists. Automatic parallelization and optimization of programs has been studied for many decades now, and a result of this is the evolution of techniques like exact data flow analysis [1] and tools like ClooG [2]. They are employed in automatic optimization and parallelization by many of today's research and commercial compilers like ICC, GCC, LLVM, XL compiler from IBM, R-Stream from Reservoir Labs, PoCC [3–6]. But the problem is by no means solved, there are several limitations on what compilers can do. For example, in order to parallelize or optimize a program, compilers need to have exact information about the data flow in the code, and this becomes hard if there are pointers in the code, so most of the above mentioned compilers limit the analysis to affine control parts in the program. Also, the techniques employed by the compilers are limited to transformations like loop unrolling, loop interchange, loop fusion, loop distribution, tiling, skewing etc. Adding to that is the complexity of today's hardware, which makes it hard to statically predict which of these transformations give better performance.

On the other hand, there is also a rise in popularity of explicit parallel programming using APIs like OpenMP, where the control is in the hands of the programmer who has the best knowledge about his/her code and tells the compiler using directives which sections of the code to parallelize. Nevertheless, if the programmer wants to try a different parallelization strategy, he/she has to change the code and also debug the parallel programs, which is as arduous as writing parallel programs itself.

The AlphaZ system developed at Colorado State University is also one such system where the user is given full control with a rich set of tools to play with and also tries to address the short comings of explicit parallel programming that were mentioned previously. AlphaZ is a compiler framework based on the polyhedral model [1, 2, 7–12] which is a mathematical formalism to reason about a class of regular computations. This kinds of computations can be commonly found in dense linear algebra kernels, computations simulating physical phenomena, in image/video/signal processing applications, in bioinformatics, optimization problems solved with dynamic programming etc. Generally, the AlphaZ users specify computations using an equational programming language called Alphabets. Alphabets programs contain systems of affine recurrence equations (SARE) defined over polyhedral domains and only specify *what* needs to be computed. Using AlphaZ, users can analyze the equations in their Alphabets programs for simplification of reductions [13], deduce parallel schedules [8] and apply affine transformations. Also, AlphaZ has a sophisticated code generation framework to develop code generators targeting various platforms and architectures. One of the code generator currently available in AlphaZ is a called Scheduled-C [14], using which users can generate OpenMP style tiled code. Along with an Alphabets program, Scheduled-C requires a target mapping specification which is a transformation specification which gives

a schedule (when), a processor allocation (where), and a memory allocation (where to store) for each computation in the Alphabets program. Additionally, users can specify tiling information in the target mapping. So if the user wishes to try a different parallelization/optimization strategy, the only thing that the user needs to specify is a new target mapping specification reflecting the strategy and call the Scheduled-C code generator. This addresses one of the concerns with writing OpenMP programs by hand.

In order for Scheduled-C to generate semantically correct code for the equations in the Alphabets program, the target map should be legal. This thesis is about a verification tool in AlphaZ that checks the legality of a given target mapping specification. In general, the Verifier is called before generating the code with Scheduled-C, thus avoiding debugging of parallel programs which was the second concern mentioned previously about writing parallel programs by hand.

The rest of this thesis is organized as follows: chapter 2 introduces the background, concepts and terminology used for rest of the thesis. Chapter 3 explains the target mapping specification in detail and also how users can specify target mapping in AlphaZ. Chapter 4 explains how the Verifier checks the legality of a given target mapping. Conclusions and future work are discussed in chapter 5.

Chapter 2

Background & Terminology

Polyhedral Model Many scientific and engineering computations spend most of their execution time in loops. The *Polyhedral model* is a mathematical framework to represent, analyze, transform and optimize a *certain class* of such computations. Many of today's research and commercial compilers [3–6] use the polyhedral model in automatic parallelization and optimization of programs. Affine control loop (ACL) programs where, the data access functions and loop bounds are affine functions of enclosing loop indices and program parameters can be readily represented in the polyhedral model. Programs with non-affine array access functions or with dynamic control can also be handled in the polyhedral model, but primarily with conservative assumptions on some of the dependences. Three out of the thirteen Berkeley motifs [15] namely "structured grids" "dense matrix" and "dynamic programming" fit in the polyhedral model and subsets from few another motifs like "graphical models" as well. In the polyhedral representation of a program, each instance of a statement is viewed as an integer point in a well defined space called the statement's polyhedron. Using exact data flow analysis [1, 16], precise information about dependences among operations in the program can be extracted and are represented as affine functions. In a bird's eve view, the polyhedral representation of a program looks like mathematical equations defined over polyhedral

domains. With such a mathematical abstraction of operations and dependences in a program, we can employ the power of linear algebra, linear programming and non-linear programming to device sophisticated static analysis and program transformations. As mathematical objects, polyhedra and affine functions enjoy a rich set of closure properties which enables us to apply these optimizing transformations repeatedly (composition of transformations) to a polyhedral specification. Across more the two decades, various techniques targeting hardware synthesis [7], memory optimization [7, 17, 18], optimization and parallelization [8, 9, 13, 19–27], code generation [2, 12, 28, 29] and program verification [30–33] have been designed for the polyhedral specifications. For more detail description of the polyhedral model please refer to [25, 34–36]

Alphabets is a equational programming language having roots in Alpha [37]. An Alphabets program defines *systems of affine recurrence equations.* In an Alphabets program, a system of affine recurrence equations hereafter referred as an affine system is given a name and a parameter domain that defines program parameters (symbolic constants) and constraints on them. An affine system will contain one or more variables and associated with the declaration of each variable is its domain, which is a polyhedral index space. Each variable in the affine system is classified as input, local or output, and each of the local and output variables is defined by exactly one equation. An equation defines values for all points in the declared domain of the corresponding variable. The expression on the RHS of an equation may be viewed as a strict single valued function whose arguments may be constants, return values of external functions, values at points in the domain of the same or other variables, reduction expressions etc. For a complete grammar of Alphabets language, please refer to [38]. The Alphabets program for forward substitution is shown below.

Listing 2.1: Alphabets program for Forward Substitution

```
affine FS_serialized {N|N-2>= 0}
given
  float L {i,k| 0<=k<i<N };
  float b {i| 0<=i<N };
returns
  float x {i| 0<=i<N };
using
  float SR_x {i,k| 0<=k<i<N};
through
  x[i] =
    case
      {|i== 0} : b[i];
      {|i>0} : (b[i] - SR_x[i,i-1]);
    esac;
  SR_x[i,k] =
    case
      {| k==0}: L[i,k] * x[k];
      {| k>0}: SR_x[i,k-1] + L[i,k] * x[k];
    esac;
.
```

Reduction is an application of an associative and commutative operator to a set of values. To express reductions in Alphabets, there is a precisely defined syntactic construct with an associated denotational and operational semantics.

In the equation

$$A[z] = \texttt{reduce}(\oplus,\ p : z' \to z,\ g(\ldots B[f(z')]\ldots)) \tag{2.1}$$

In the expression $\texttt{reduce}(\oplus,\ p : z' \to z,\ g(\ldots B[f(z')]\ldots))$

- $\oplus$ *is a binary operator which is both associative and commutative.*
- $p : z' \to z$ and $p(z') = z$ is called as *projection function*, and in almost all cases is a *many-to-one* mapping.
- $g(\ldots B[f(z')]\ldots)$ is called the *reduction body* and is defined for all points in the domain $\mathcal{D}_{\text{body}}$ where

$$\mathcal{D}_{\text{body}} = \{z' | z' \in D_E\ ;\ p(z') = z\ ;\ z \in D_A\}$$

where,

- D_E is domain of the expression $g(\ldots B[f(z')]\ldots)$.
- D_A is the domain of the variable A.

So for each point z in the domain of A, to produce the value at $A[z]$, the expression $g(\ldots B[f(z')]\ldots))$ is evaluated for a set of points $\{z'|z' \in \mathcal{D}_{\text{body}}\ ;\ p(z') = z\}$ and reduced by applying the operator $\oplus$.

Example: With reductions, the above forward substitution computation can be expressed more compactly as shown below.

Listing 2.2: Alphabets program with reductions for Forward Substitution

```
affine FS_reduce {N|N-2>= 0}
given
  float L {i,k| 0<=k<i<N };
  float b {i| 0<=i<N };
returns
  float x {i| 0<=i<N };
using

through
  x[i] = case
      {|i== 0} : b[i];
      {|i>0} : (b[i] - reduce(+, (i,k->i), L[i,k] * x[k]));
    esac;
.
```

AlphaZ AlphaZ is a compiler framework to analyse, transform and generate code for polyhedral specifications. AlphaZ has a simple command script interface and is integrated into Eclipse IDE. AlphaZ can read Alphabets programs and do the following

- Check for the semantic correctness of the Alphabets program
- Analyse for schedules.
- Simplify reductions in the program.

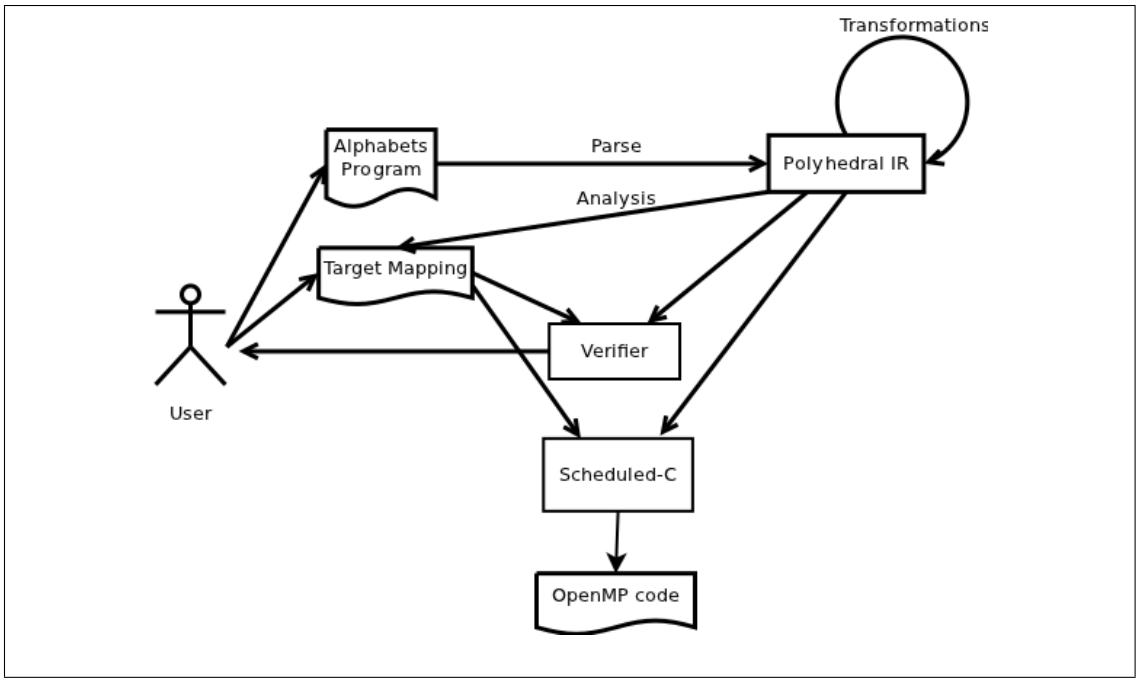

Figure 2.1: AlphaZ block diagram highlighting the Verifier and the Scheduled-C code generator

- Apply optimizing transformations to the program
- Generate code for the equations in the program.

Currently there are two code generators available in AlphaZ, first is a "memoizated demand driven" sequential code generator called Write-C and the other is called Scheduled-C which can generate tiled OpenMP style parallel code. In order to use Scheduled-C, along with the Alphabets program, users have to specify a target-mapping specification for the Alphabets program. The Verifier checks the legality of the target-mapping specification for the Alphabets program. The block diagram of the AlphaZ system highlighting the Verifier and the Scheduled-C code generator is shown in the figure 2.1. For the rest of this thesis, we will use the term code generator to refer to Scheduled-C.

Chapter 3

Target Mapping

Target mapping for an Alphabets program is a term that collectively refers to

- Space-time mapping.
- Statement ordering information.
- Memory mapping.
- Memory space sharing.
- Tiling specification.

This along with the Alphabets program is input to the Verifier whose job is to check the legality of the target mapping for the Alphabets program. In this chapter, we will discuss in detail, each component of the target mapping and how to specify a target mapping for an Alphabets program.

3.1 Space-Time Mapping

A space-time mapping is an *affine function* that maps each point in the domain of a variable to a *time* instance and *space (processor)* on which the value at that point should be computed. Additionally, with extra dimensions, statement ordering

information which specifies the textual order in which statements and loops should be placed in the code can also be specified in the space-time mapping.

The code generator uses space-time mapping information in the pre-processing and the post-processing steps of the code generation. In the pre-processing step, *space* and *time* dimensions of the space-time mapping are applied as *change-of-basis transformations* before calling ClooG [2] to generate loops. The *depth of the loop nest* in the generated code will be equal to the number of space and time dimensions in the space-time mapping. In the post-processing step, the loops corresponding to the space dimensions in the space-time mapping will be *parallelized* using `#pragam omp parallel` construct and the statement ordering information is used while embedding statements inside the generated loops.

For rest of this thesis, ϕ will be used to refer to a space-time mapping in general. $\phi\{$*subscript:variable name*$\}$ will refer to the space-time mapping of variable. If ϕ_A is the space-time mapping of a variable named A then ϕ_A^i is a row in ϕ_A and $0 \leq i < m$, where m is the number of dimensions in the space-time mapping. The term ***Schedule*** will be used interchangeably with space-time mapping.

3.1.1 Specifying the Space-Time Mapping

In AlphaZ, space-time mapping for a variable is specified using the command

```
setSpaceTimeMap(Program program, String affineSystem, String
   variable, String spaceTimeMap);
```

Listing 3.1: Alphabets program for Optimal String Parenthesization

```
affine MCM_serialized {N| N-2>= 0}
given
  unsigned int A {i| N-i>= 0 && i>= 0};
returns
  unsigned int min_cost {|};
using
  unsigned int M {i,j| 1<=i<=j<=N};
  unsigned int SR_M {i,j,k| 1<=i<=k<j<=N};
through

  M[i,j] =
    case
    {|j==i} : 0;
    {|j>i} : SR_M[i,j,j-1];
    esac;

  min_cost[] = M[1,N];

  SR_M[i,j,k] =
    case
      {| k==i}: M[i,k] + M[k+1,j] + A[i-1] * A[k] * A[j];
      {| k>i}: min(SR_M[i,j,k-1],
          M[i,k] + M[k+1,j] + A[i-1] * A[k] * A[j]);
    esac;
.
```

Example: Consider the Alphabets program for optimal string parenthesization (matrix chain multiplication) in listing 3.1. Given a sequence of matrices, the program finds the cost (total number of operations) that will be incurred when the matrices are multiplied in the most efficient order. Input to the program is an array `A` containing the sizes of the matrices and output of the program is the value `min_cost`. Each $\langle i, j \rangle$ in the table `M` gives the minimum cost that will be incurred to multiply the sequence from i to j. So the final answer for the sequence of N matrices will be stored in `M[1,N]`. For each `M[i,j]` the answer is the minimum

of M[i,k] + M[k+1,j]+ the cost of multiplying the partial sequences M[i,k] and M[k+1,j] which is A[i-1] ∗ A[k] ∗ A[j] and k is $i \leq k \leq j$. In the program, the reduction needed for each M[i,j] is serialized using the variable SR_M. An example of a space-time mapping specification for this program is

```
setSpaceTimeMap(prog, "MCM_serialized", "SR_M",
                      "(i,j,k → j-i,i,k)");
setSpaceTimeMap(prog, "MCM_serialized", "M",
                      "(i,j → j-i,i,j)");
setSpaceTimeMap(prog, "MCM_serialized", "min_cost",
                      "( → N-1,1,N+1)");
```

Notice that, space-time mapping is specified only for *local* and *output* variables in the Alphabets program. In AlphaZ, specifying space-time mapping for input variables is optional. In case a space-time mapping is not specified for any of the *input* variables, then the Verifier assumes that the values at all points in the domain of that variable are available before the beginning any computation.

3.1.2 Specifying Parallel Schedules

By default all dimensions in a space-time mapping will be sequential. To specify parallel schedules and statement ordering information Each dimension of a space-time mapping can be *annotated* as SEQUENTIAL or PARALLEL or ORDERING using the command

```
setDimensionType(Program program, String affineSystem, String
   variable, int dimension, String type)
```

A parallel schedule is a space-time mapping specification in which at least one of the dimensions is annotated as PARALLEL. There is no restriction on which dimensions can be annotated as PARALLEL.

Example: For the Alphabets program for optimal string parenthesization in listing 3.1, the space-time mapping specification with the second dimension annotated as parallel is

```
setSpaceTimeMap(prog, "MCM_serialized", "SR_M",
                      "(i,j,k → j-i,i,k)");
setSpaceTimeMap(prog, "MCM_serialized", "M",
                      "(i,j → j-i,i,j)");
setSpaceTimeMap(prog, "MCM_serialized", "min_cost",
                      "(  → N-1,1,N+1)");

setDimensionType(prog, "MCM_serialized", "SR_M", 1, "PARALLEL");
setDimensionType(prog, "MCM_serialized", "M", 1, "PARALLEL");
setDimensionType(prog, "MCM_serialized", "min_cost", 1,
                          "PARALLEL");
```

In the code generated with the above space-time mapping specification, the second loop in the loop nest will be parallelized with an `#pragma omp parallel for` construct.

3.1.3 Statement Ordering

Statement ordering is the information given to the code generator about the textual order in which loops or statements should be placed inside the code [9, 39, 40].

Ordering information can also be treated as an extension to the notion of *time*, which till now only refers to the sequential components of the iteration vectors of the loop nest.

```
for (int i = 1; i < N; ++i) {
  A[i] = ...
  B[i] = A[i] + ...;
}
```

For example, in the above code snippet, even though each value of A read by B comes from the same iteration of the loop, because the statement for A appears before statement for B in the loop body, *the operation* $\langle A, i\rangle$ *will execute before the operation* $\langle B, i\rangle$.

In AlphaZ, ordering information can be specified in two ways

- with additional dimensions in the space-time mapping.
- or by using the command `setStatementOrdering`.

3.1.3.1 Statement Ordering with ORDERING Dimensions

Statement ordering can be specified by adding extra dimensions in the space-time mapping specification and annotating those dimensions as `ORDERING`. The ordering information should always be a *non-negative integer* denoting the position in which the loops or statements corresponding to the variable should appear in the code. Since the ordering is relative, i.e., with respect to other variables, The dimension that are annotated as `ORDERING` should be *common* across all the space-time mappings in a given space-time mapping specification.

Example: For the Alphabets program for optimal string parenthesization in listing 3.1, space-time mapping specification with statement ordering information is

```
setSpaceTimeMap(prog, system, "SR_M", "(i,j,k → 0,j-i,i,0,k)");
setSpaceTimeMap(prog, system, "M", "(i,j → 0,j-i,i,1,0)");
setSpaceTimeMap(prog, system, "min_cost", "( → 1,0,0,0,0)");

setDimensionType(prog, system, "SR_M", 0, "ORDERING");
setDimensionType(prog, system, "M", 0, "ORDERING");
setDimensionType(prog, system, "min_cost", 0, "ORDERING");

setDimensionType(prog, system, "SR_M", 3, "ORDERING");
setDimensionType(prog, system, "M", 3, "ORDERING");
setDimensionType(prog, system, "min_cost", 3, "ORDERING");
```

In the above space-time mapping specification, the first and the forth dimensions are annotated as `ORDERING`. The ordering information for variable `min_cost` differs from that of the other two variables in the first dimension. In the forth dimension, the variables `SR_M` and `M` have different ordering information. The code generated by Scheduled-C with the above space-time mapping specification will something like the following

```
for(c0 ...)
  for(c1...) {
    for(c2...) {
      SR_M[c0, c1, c2] =
    }
    M[c0, c1] =
  }
}
min_cost[] =
```

3.1.3.2 Specifying Ordering Information with setStatementOrdering

```
setStatementOrdering(Program program, String affineSystem,
    String predecessorEquation, String successorEquation);
```

If the ordering information is specified using the command `setStatementOrdering`, then it always refers to the order in the which statements relating to variables `predecessorEquation` and `successorEquation` should appear inside the *innermost* loop common to both the variables.

3.1.4 Space-Time Mapping for Reductions

In AlphaZ, a space-time mapping can be specified for the reduction expressions in the Alphabets programs, and it can be either for the *reduction body* or for the *reduction result*.

Consider the following reduction equation

$$A[z] = \ldots \texttt{reduce}(\oplus,\ p : z' \rightarrow z,\ g(\ldots B[f(z')]\ldots))\ldots \tag{3.1}$$

In the above equation, let D_A be the domain of variable A and $\mathcal{D}_{\text{body}}$ the context domain of the reduction expression. For each point z in the D_A, there is a set of points $\{z' | z' \in \mathcal{D}_{\text{body}}\ ;\ p(z') = z\}$ for which the expression $g(\ldots Y[f(z')]\ldots)$ is evaluated and combined by applying the operator $\oplus$ to produce the value at $A[z]$.

3.1.4.1 Space-Time Mapping for Reduction Body

When the space-time mapping is specified for the reduction body, for each point z' in $\mathcal{D}_{\text{body}}$, it is the time and processor on which expression $g(\ldots Y[f(z')]\ldots)$ is evaluated. Its time component can be viewed as an order in which expression $g(\ldots Y[f(z')]\ldots)$ should be evaluated for each point z' in $\mathcal{D}_{\text{body}}$, and this order depends on the availability of values needed to evaluate the expression $g(\ldots Y[f(z')]\ldots)$.

Listing 3.2: Alphabets program for optimal string parenthesization with reductions

```
affine MCM_normalized {N| N-2>= 0}
given
  unsigned int A {i| N-i>= 0 && i>= 0};
returns
  unsigned int min_cost {|};
using
  unsigned int M {i,j| 1<=i<=j<=N};
  unsigned int NR_M {i,j| 1<=i<j<=N};
through

  M[i,j] =
    case
      {|-i+j== 0} : 0;
      {|-i+j-1>= 0} : NR_M[i,j];
    esac;

  min_cost[] = M[1,N];
  NR_M[i,j] = reduce(min, (i,j,k->i,j), {|i<=k<j} :
      ((M[i,k] + M[k+1,j]) + ((A[i-1] * A[k]) * A[j])));
.
```

Example: For the optimal string parenthesization program in listing 3.2, an example of a space-time mapping specification with space-time mapping for the reduction body in the equation for `NR_M` is

```
    setSpaceTimeMap(prog, system, "NR_M",  "(i,j,k → j-i,i,k)");
    setSpaceTimeMap(prog, system, "M",  "(i,j → j-i,i,j)");
    setSpaceTimeMap(prog, system, "min_cost",  "( → N-1,1,N+1)");
```

3.1.4.2 Space-Time Mapping for Reduction Result

When the space-time mapping is specified for the reduction result, for each point z in D_A, it is the time and processor on which the *result* of applying $\oplus$ to the values of the expression $g(\ldots B[f(z')]\ldots)$ for all points in the set $\{z'|z' \in \mathcal{D}_{R_A}\ ;\ p(z') = z\}$ is computed. Note that since the computation of this result may require a large number of values, the code generator generates a sequential loop to compute and accumulate all these values. Therefore the interpretation of a "time step" is not that of a constant duration, rather its the time to execute a (possible nested) loop. It is the verifier's job to ensure the legality of such a schedule.

Example: For the optimal string parenthesization program in listing 3.2, an example of a space-time mapping specification with space-time mapping for the reduction result in the equation for `NR_M` is

```
setSpaceTimeMap(prog, system, "NR_M",  "(i,j → j-i,i,0)");
setSpaceTimeMap(prog, system, "M",  "(i,j → j-i,i,1)");
setSpaceTimeMap(prog, system, "min_cost",  "( → N-1,1,2)");
```

Observe that for either case (reduction body or reduction result), the third argument to `setSpaceTimeMap` is the name of the variable on the left hand side of the reduction equation. The only way to distinguish if the space-time mapping is intended for the reduction body or the reduction result is with the number of input indices in the space-time mapping. If the space-time mapping is for the reduction body then, the number of input indices will be equal to the number of dimensions in the context domain of the reduction body. If the space-time mapping is for the reduction result then, the number of indices will be equal to the number of dimensions in the domain of the reduction variable.

3.2 Memory Mapping

For a variable in an Alphabets program, a memory mapping assigns points in the domain of a variable to memory locations. The code generator uses the memory mapping specification to allocate memory for all the variables and also to create macros which map iteration vectors to array indices. By default, each point in the domain of a variable will write into a unique memory location. So the code generator will allocate memory for each variable according to the size of its domain, this may be usually inefficient. For example, for the case of matrix multiplication, all that might be needed is a single scalar variable to accumulate $A[i][k] * B[k][j]$ for each $C[i][j]$. Indeed, we may even directly accumulate the partial results into the output variable C. However, since Alphabets is a functional language, the temporary accumulation variable for the matrix multiplication is declared as a 3-dimensional variable with $\Theta(N^3)$ points in its domain. With the default memory allocation, the code generator will allocate $\Theta(N^3)$ memory. To avoid this, users may specify many-to-one functions as memory mappings.

In AlphaZ, a memory mapping is specified using the command

```
setMemoryMap( Program program, String system, String var,
              String memoryMap);
```

Example: For the optimal string parenthesization Alphabets program in listing 3.1, an example of a memory mapping for the variable `SR_M` can be

```
setMemoryMap(program, "MCM_serialized", "SR_M",
                      "(i,j,k → i,j)");
```

Here, for each $\langle i, j \rangle$ of `SR_M`, the set of points
$\{k \mid i \leq k < j\}$ will write into a same memory location. Along with the iteration space for the program, the memory foot print for the variables `SR_M` and `M` are shown (in dotted lines at the bottom) in the figure 3.1.

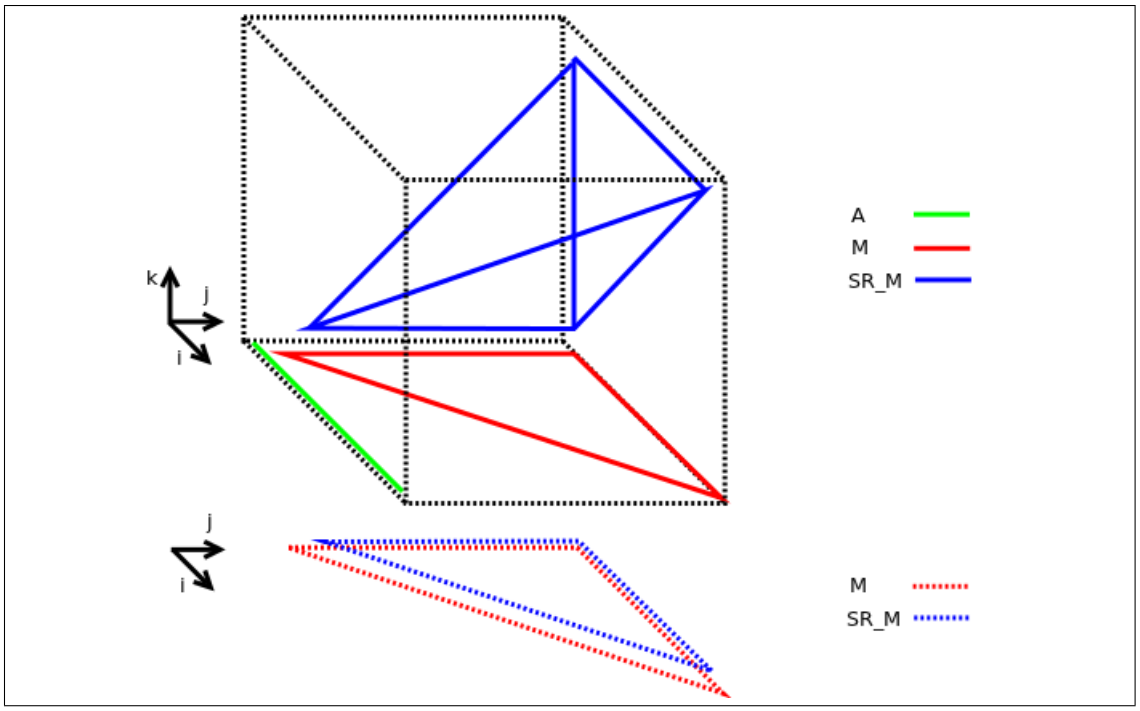

Figure 3.1: For the optimal string parenthesization Alphabets program in listing 3.1, the iteration space and the memory foot print if the memory mapping for the variables `SR_M` and `M` are $(i, j, k \rightarrow i, j)$ and $(i, j \rightarrow i, j)$ respectively.

3.3 Memory Space Sharing

In AlphaZ, we can specify multiple variables in an Alphabets program to share memory in the generated code.

Listing 3.3: Alphabets program for optimal string parenthesization with reductions

```
affine MCM_reduce {N| N-2>= 0}
given
  unsigned int pA {i| 0<=i<= N};
returns
  unsigned int min_cost {|};
using
  unsigned int pM {i,j| 1<=i<=j<=N };
through

    pM[i, j] =
    case
      {| i == j}: 0;
          {| i < j}: reduce(min, [k], {| i <= k < j}:
     pM[i, k] + pM[k+1, j] + (pA[i-1] * pA[k] * pA[j]));
      esac;

    min_cost[] = pM[1, N];
.
```

To motivate memory space sharing, a version of optimal string parenthesization Alphabets program with reductions is listed in 3.3. The difference between this and the earlier version is that the old version has the reduction *serialized* by using an extra variable `SR_M`. If we generate code for the serialized program, by default the code generator will allocate $O(N^3)$ memory for the variable `SR_M`, where N is the program parameter in the Alphabets program.

For a class of space-time mapping specifications, there may not be a need to allocate memory for the variable `SR_M` in the generated code, instead `SR_M` may share memory space with the variable `M`.

In AlphaZ, a memory space sharing is specified using the command

```
setMemoryMap( Program program, String system, String var,
   String memorySpace, String memoryMap);
```

Variables sharing common memory space should have the same value for the parameter `memorySpace`.

Example: For the Alphabets program for optimal string parenthesization in listing 3.1, an example of a target mapping specification with variables `M` and `SR_M` sharing the same memory space is

```
setSpaceTimeMap(prog, "MCM_serialized", "SR_M",
                      "(i,j,k→j-i,i,k)");
setSpaceTimeMap(prog, "MCM_serialized", "M",
                      "(i,j → j-i,i,j)");
setSpaceTimeMap(prog, "MCM_serialized", "min_cost",
                      "(  → N-1,1,N+1)");

  setMemoryMap(program, "MCM_serialized", "SR_M", "M",
                        "(i,j,k → i,j)");
  setMemoryMap(program, "MCM_serialized", "M", "M",
                        "(i,j → i,j)");
```

The iteration space for the Alphabets program for optimal string parenthesization in listing 3.1 and and memory foot print according to the above target mapping specification is shown in the figure 3.2.

3.4 Tiling Specification

Tiling is a loop optimization technique where the points in an iteration space are grouped into smaller blocks (tiles) for coarse grain parallelization or to enhance data locality. In the case of enhancing data locality, tiling is done to take advantage of memory hierarchy so that data needed for execution of a tile can fit into caches or registers and allows reuse. When tiled for coarse grain parallelism, all the points in a tile can be computed on a single processor, and multiple tiles can execute concurrently with inter processor communication required only before or after the execution of a tile.

In AlphaZ, tiling is specified using the command

```
setTiling(program, system, dimension, tile_size);
```

The parameter `dimension` in the command specifies which dimension in the space-time mapping specification has to be tiled, and `tile_size` parameter spec-

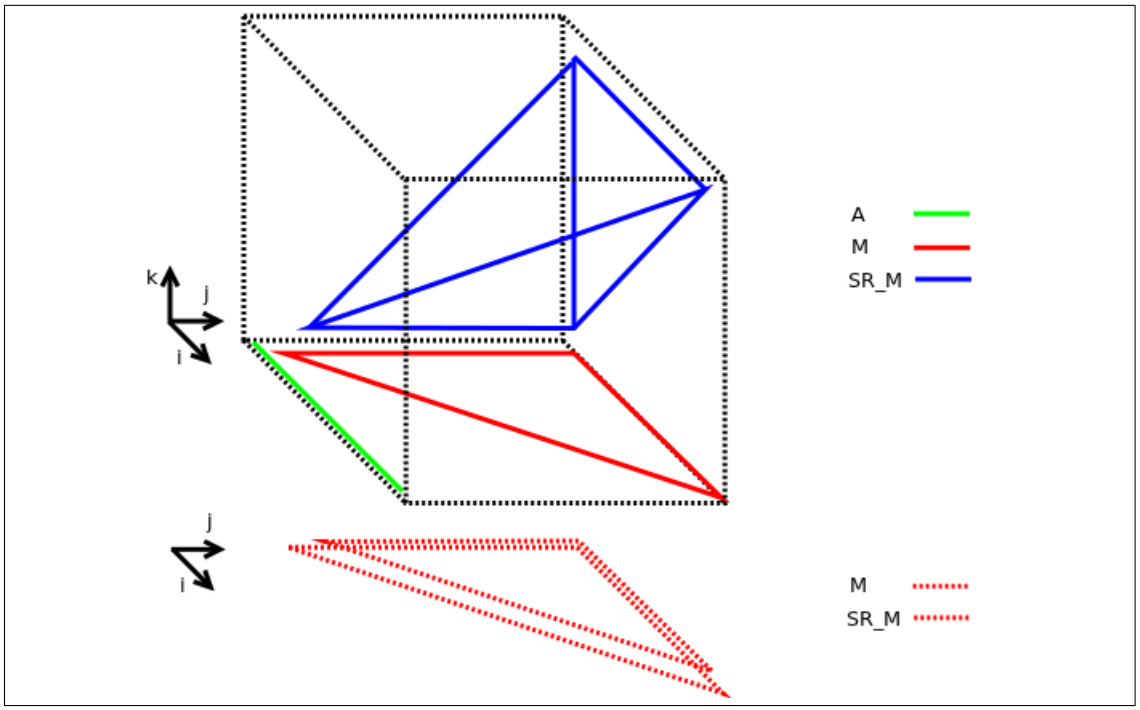

Figure 3.2: For the optimal string parenthesization Alphabets program in listing 3.1, the iteration space and the memory foot print if the memory mapping for the variables `SR_M` and `M` are $(i, j, k \rightarrow i, j)$ and $(i, j \rightarrow i, j)$ respectively and both `SR_M` and `M` share the same memory space.

ifies the size of the tile. The code generator uses a tool called D-tiler [24] to tile the loops that it gets from ClooG [2].

Listing 3.4: Alphabets program for Matrix Multiplication

```
affine MM_serialized {P, Q, R|P>0 && Q>0 && R>0}
given
  float A {i,k| 0<=i<P && 0<=k<Q};
  float B {k,j| 0<=k<Q && 0<=j<R};
returns
  float C {i,j| 0<=i<P && 0<=j<R};
using
   float Acc_C {i,j,k| 0<=i<P && 0<=j<R && 0<=k<=Q};
through
  Acc_C[i,j,k] =
      case
            {|k==0} : 0;
          {|k>0}  : Acc_C[i,j,k-1] + A[i,k-1]*B[k-1,j];
        esac;
   C[i,j] = Acc_C[i,j,Q];
.
```

Example: For the Alphabets program for matrix multiplication above, an example of a tiling specification telling the code generator to tile all the three dimensions in the generated program is

```
  setSpaceTimeMap(prog, "MM_serialized", "Acc_C",
                        "(i,j,k → k,i,j)");
  setSpaceTimeMap(prog, "MM_serialized", "C",
                        "(i,j → Q+1,i,j)");

  setTiling(program, system, 0, 30);
  setTiling(program, system, 1, 30);
  setTiling(program, system, 2, 30);
```

Chapter 4

Legality

Inputs to the Verifier are an Alphabets program and a target mapping specification. Based on the information about the *dependences* in the Alphabets program, the Verifier checks the legality of the proposed target mapping. The target map for the Alphabets program is said to be legal *iff* each of the components of the target map, i.e., space-time mapping, memory mapping and tiling specification are legal.

In this chapter, we will discuss in-detail the legality conditions for these three components. In order to do this, we will define a precedence relation operator that specifies when an operation precedes another in a loop program. Specifically, we define this operator for operations in an OpenMP program, since this is the target for the code generator.

Execution Order

The code generator generates code in which all the statements will be surrounded by an equal number of loops. The D-Tiler [24] tool that the code generator uses to tile the loops requires the code to be in this style. The precedence relation operator which we will now define will be based on operations in programs where loops obey this rule. Nevertheless, the precedence relation can be trivially extended to operations in loops in which different statements may be surrounded by different

number of loops. Note that the convention of embedding all the statements in the same number of dimensions does not introduce any loss of generality, either for the code generator, or for the Verifier.

Because of the above convention, in the loops produced by the code generator, the iteration vectors in the domain of every statement will have the same number of indices. For the sake of simplicity and technical correctness, we will henceforth *extend* the iteration vectors to include constants defining the textual order of the statements. If we include statement ordering information at every loop level, then the size of the iteration vectors will be $2k+1$, where k is the depth of the loop nest. If there is only one statement or none at a particular loop level, then the statement ordering information at that level will be of no significance so the corresponding component of iteration vector will be dropped.

Consider a sequential loop nest with two statements S_1 and S_2 and let z_1 and z_2 be two iteration vectors in the domains of statements S_1 and S_2 respectively.

For a sequential program, the operation $\langle S_2, z_2 \rangle$ executes after the operation $\langle S_1, z_1 \rangle$, which is written as $\langle S_2, z_2 \rangle \gg \langle S_1, z_1 \rangle$ *iff* $z_2 \succ z_1$, where, $\succ$ is *strict lexicographic* order.

Note that this definition for execution order only applies to sequential loop nests in which the order of execution is according to the lexicographic order of iteration vectors. If there are parallel loops in the loop nest then, we need a new set of rules to determine the execution order of operations. We motivate this with an example with dependences and then define the rules for the execution order.

Consider a loop nest as follows

```
  for(int i = 1; i <= N; ++i) {
    for(int j = 1; j <= i; ++j) {
S1:     B[i][j] = ...
S2:     A[i][j] = B[i-1][j] + A[i][j-1] + B[i][j]
    }
  }
```

In this loop nest, to compute the value for `A[i][j]` in statement S_2, operation $\langle S_2, i, j, 1\rangle$ requires the values of `B[i-1][j]`, `A[i][j-1]` and `B[i][j]`.

The operations which have produced these values are

- $\langle S_1, i-1, j, 0\rangle$, for the value at `B[i-1][j]`.
- $\langle S_2, i, j-1, 1\rangle$, for the value at `A[i][j-1]`.
- $\langle S_1, i, j, 0\rangle$, for the value at `B[i][j]`.

Since the loop executes according to the lexicographic order of the iteration vectors, the order of execution will be

- $\langle S_2, i, j, 1\rangle \gg \langle S_1, i-1, j, 0\rangle$.
- $\langle S_2, i, j, 1\rangle \gg \langle S_2, i, j-1, 1\rangle$.
- $\langle S_2, i, j, 1\rangle \gg \langle S_1, i, j, 0\rangle$.

Now, if the outer loop is *parallelized*, then the execution order will no longer be according to the lexicographic order of iteration vectors. Now each iteration of the outer loop i can execute independently. Hence, there is no guarantee on the availability of some of values required to compute `A[i][j]`. Precisely, it is the operation $\langle S_1, i-1, j, 0\rangle$, which produces the value of `B[i-1][j]` may now be computed on a different processor concurrently. So

- $\langle S_2, i, j, 1\rangle \gg \langle S_1, i-1, j, 0\rangle$ may not be true.

For the operations $\langle S_2, i, j-1, 1\rangle$ and $\langle S_1, i, j, 0\rangle$ which produce the values `A[i][j-1]` and `B[i][j]` respectively, the condition $\langle S_2, i, j, 1\rangle \gg \langle S_2, i, j-1, 1\rangle$, $\langle S_2, i, j, 1\rangle \gg \langle S_1, i, j, 0\rangle$ still holds good as the two operations are mapped on to the same processor i as $\langle S_2, i, j, 1\rangle$ and the inner loop still executes in the lexicographic order of iteration vectors.

Consider yet another situation in which instead of the outer loop i, if the inner loop j is parallelized, because the lexicographic order condition is satisfied in the loops prior to the parallel loop i, $\langle S_2, i, j, 1\rangle \gg \langle S_1, i-1, j, 0\rangle$. But now there is no guarantee on the operation $\langle S_2, i, j-1, 1\rangle$ which produces the value `A[i][j-1]`. However the condition $\langle S_2, i, j, 1\rangle \gg \langle S_1, i, j, 0\rangle$ still holds good as the dependence is not carried by the loop indices, rather it is based on the textual order of statements S_1 and S_2.

Based on the above observations, we can now say that for any two operations $\langle S_2, z_2\rangle$ and $\langle S_1, z_1\rangle$ in a parallel loop nest, $z_2 \succ z_1$ does not guarantee $\langle S_2, z_2\rangle \gg \langle S_1, z_1\rangle$. The comparison of two operations for execution order should not be completely based lexicographic order of iteration vectors, rather we now need to consider the *loop types* while comparing the components of the iteration vectors. We now define the execution order of operations in a parallel loop nest.

We say $\langle S_2, z_2\rangle \gg \langle S_1, z_1\rangle$ if either of the following conditions are satisfied

- In the loop nest, if dimension i is parallel, then $z_2 \succ_{i-1} z_1$. i.e., $z_2 \succ z_1$ should be satisfied in the first $i-1$ components of the two vectors z_2 and z_1.

- In the loop nest, if dimension i is parallel and $z_2 =_{i-1} z_1$ then $z_2^i = z_1^i$ and $z_2 \succ_{i+1\ldots n} z_1$. i.e., z_2^i should be equal to z_1^i and $z_2 \succ z_1$ should be satisfied in the components $\{i+1 \ldots n\}$ of the two vector, where n is the size of the vectors.

For the rest of this thesis, we will use $\rhd$ and $\trianglerighteq$ to describe *the precedence relationship between two iteration vectors in an OpenMP program* based on the rules that we have defined above.

4.1 Legality of Space-time Mapping

To generate code for an Alphabets program, the code generator requires a target mapping specification. In the process of code generation, first, the code generator uses the space-time mappings in the target mapping specification as *change of basis* functions and transforms the Alphabets program. This transformation brings all the variables in the Alphabets program into a common space. In the next step, it uses ClooG [2] to generate loops to visit all the integer points in the union of the domains of the equations in this transformed Alphabets program. Then, the code generator post-processes the generated loops and calls D-tiler [24] to tile the dimensions indicated in the target mapping. Next, based on the space-time mapping, the code generator annotates the loops as parallel. In the final step, the code generator embeds statements relating to equations in the loops.

In the generated code, the code generator guarantees that every computation in the Alphabets program corresponds to an operation in the generated code. i.e., every point in the domain of every local/output variable is executed as an instance of some statement in the generated code. Also it guarantees that the order of execution of operations in the code respects the space-time mapping specification.

However, because ClooG may *split* the domains as it generates loops, the iteration vectors of the statements instances in the generated code may be *extensions* to the vectors obtained by applying the space-time mapping to the points in the domains of the variables in the Alphabets program. Nevertheless, the additional dimensions in the iteration vectors of the generated code introduced because of splitting are artifacts of the code generator and are not available to the Verifier. So the Verifier only works with the vectors that it obtains by applying the space-time mapping to the points in the domains of the variables in the Alphabets program.

Example 1 Consider an Alphabets program with two variables `A` and `B` with domains D_A and D_B respectively and equations of the form

$$B[z'] = \ldots$$
$$A[z] = g(\ldots B[f(z)] \ldots)$$

Given a space-time mapping specification for this Alphabets program, in which ϕ_A is the space-time map for variable `A` and ϕ_B is the space-time map for variable `B`. Then, for two points z and z' in D_A and D_B respectively, $\phi_A(z)$ and $\phi_B(z')$ are the iteration vectors at which the operations $\langle A, z\rangle$ and $\langle B, z'\rangle$ will execute. If $\phi_A(z) \rhd \phi_B(z')$, then $\langle A, z\rangle \gg \langle B, z'\rangle$.

4.1.1 Legality Condition

The Verifier checks the legality of space-time mapping based on the dependences in the Alphabets program. For each dependence in the Alphabets program, the Verifier checks if the execution order precedence relationship between the producer and consumer will be respected in the generated code. *The Verifier terms a dependence as satisfied, iff for all the points in the domain of the dependence, the producer precedes the consumer in the execution order.*

If all the dependences in the Alphabets program are satisfied, then the Verifier terms the space-time mapping specification as legal. Prior to checking if the dependencies in the program are satisfied, the verifier makes sure that all the space-time mappings in the target mapping specification are *bijection* functions.

In an Alphabets program, let v denote a variable defined over a domain D_v and V denote the set of all the variables in the program, i.e, $v \in V$. Let e denote a dependence in the Alphabets program and C_e and P_e be the consumer and producer of e, and $C_e, P_e \in V$. Let f_e denote the dependence function and $\mathcal{D}_e$ be

the context domain of e. For the rest of this thesis, we will write the dependence e as

$$\forall z \in \mathcal{D}_e : \langle C_e,\ z\rangle \rightarrow \langle P_e,\ f_e(z)\rangle$$

Let E denote the set of all the dependences in the Alphabets program.

Given a space-time mapping specification Φ for the program, where ϕ_v denotes space-time mapping for variable v and $v \in V$. The Verifier terms the space-time mapping specification Φ as legal *iff* $\forall e \in E$

$$\forall z \in \mathcal{D}_e : \langle C_e,\ z\rangle \gg \langle P_e,\ f_e(z)\rangle$$

which is equivalent to

$$\forall z \in \mathcal{D}_e : \phi_{C_e}(z) \rhd \phi_{P_e}(f_e(z))$$

Example: Consider the Alphabets program for optimal string parenthesization (matrix chain multiplication) shown in the figure 3.1. The domains of the variables `A`, `M`, `SR_M` are shown in the figure 4.1 along with some of the dependences in the equation for the variable `SR_M`.

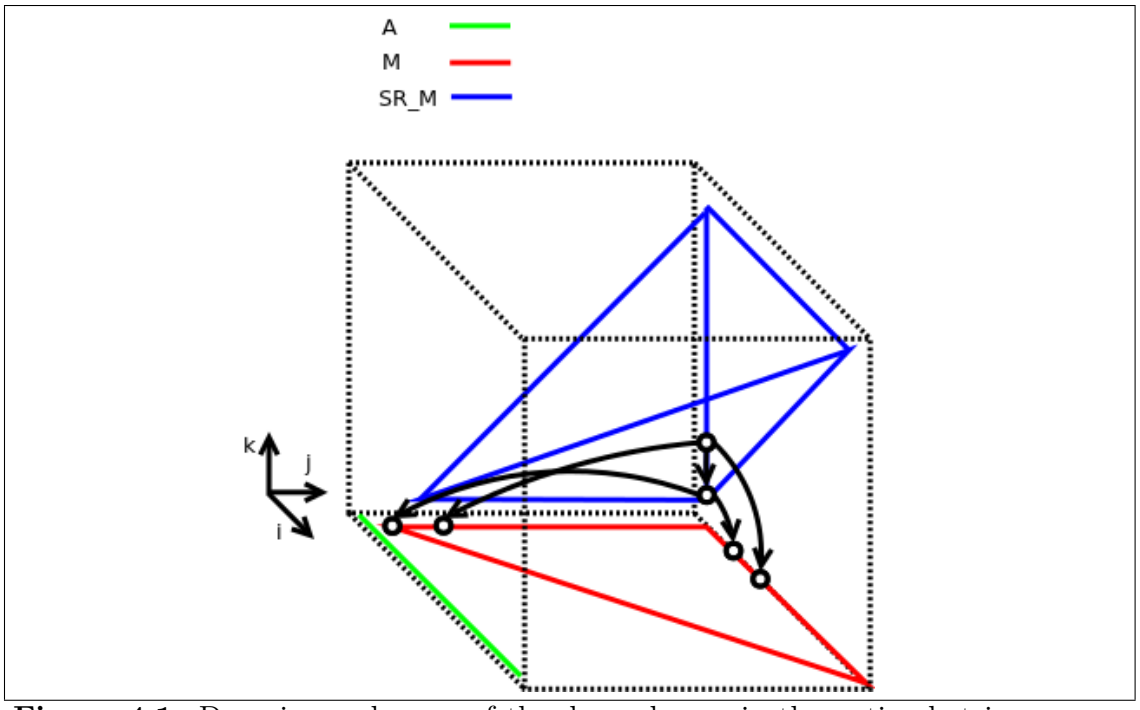

Figure 4.1: Domains and some of the dependences in the optimal string paranthesization program in listing: 3.1

For this program, consider a space-time mapping specification as follows

```
setSpaceTimeMap(prog, "MCM_serialized", "SR_M",
                              "(i,j,k → j-i,i,k)");
setSpaceTimeMap(prog, "MCM_serialized", "M",
                              "(i,j → j-i,i,j)");
setSpaceTimeMap(prog, "MCM_serialized", "min_cost",
                              "( → N-1,1,N+1)");
```

Let us see the dependence satisfiability condition for some of the dependences, namely $\forall\langle i,j,k\rangle \in \mathcal{D}_{SR\_M} : \texttt{SR\_M}(i,j,k) \rightarrow \texttt{M}(i,k)$ and $\forall\langle i,j,k\rangle \in \mathcal{D}_{SR\_M} : \texttt{SR\_M}(i,j,k) \rightarrow \texttt{M}(k+1,j)$ from the equation for SR_M. The context domain $\mathcal{D}_{SR\_M}$ for both the dependences is $\{i,j,k|1 \leq i \leq k < j\}$.

The condition for the dependence $\forall\langle i,j,k\rangle \in \mathcal{D}_{SR\_M} : \texttt{SR\_M}(i,j,k) \rightarrow \texttt{M}(i,k)$ to be satisfied is

$$\forall\langle i,j,k\rangle \in \mathcal{D}_{SR\_M} : \phi_{SR\_M}(i,j,k) \rhd \phi_M(i,k)$$

Similarly, the condition for the dependence

$\forall\langle i,j,k\rangle \in \mathcal{D}_{SR\_M} : \texttt{SR\_M}(i,j,k) \rightarrow \texttt{M}(k+1,j)$ to be satisfied is

$$\forall\langle i,j,k\rangle \in \mathcal{D}_{SR\_M} : \phi_{SR\_M}(i,j,k) \rhd \phi_M(k+1,j)$$

The Verifier goes through dimension by dimension of the space-time map to check if the dependences are satisfied. For the above two dependences, the condition the Verifier checks in the first dimension of the space-time mapping are

For the dependence $\forall\langle i,j,k\rangle \in \mathcal{D}_{SR\_M} : \texttt{SR\_M}(i,j,k) \rightarrow \texttt{M}(i,k)$

$$\begin{aligned}\forall\langle i,j,k\rangle \in \mathcal{D}_{SR\_M} : j - i &> k - i \\ &\equiv j - k > 0\end{aligned}$$

For the dependence $\forall\langle i,j,k\rangle \in \mathcal{D}_{SR\_M} : \texttt{SR\_M}(i,j,k) \rightarrow \texttt{M}(k+1,j)$

$$\begin{aligned}\forall\langle i,j,k\rangle \in \mathcal{D}_{SR\_M} : j - i &> j - (k+1) \\ &\equiv k - i > -1\end{aligned}$$

In order to check if affine forms like $j - k > 0$ and $k - i > -1$ to be true for all points in $\mathcal{D}_{SR\_M}$, the Verifier reduces the problems to *emptiness of polyhedra check.*

Emptiness of Polyhedra Check *An affine form $\psi(z)$ is non-negative i.e., $\psi(z) \geq 0$, for all points z in a domain $\mathcal{D}$ iff the negation of $\psi(z)$ i.e., $\psi(z) < 0$ is not true for any point z in $\mathcal{D}$.* This can be checked by adding $\psi(z) < 0$ as another affine inequality to the domain $\mathcal{D}$. Let us denote the new domain $\mathcal{D} + \{\psi(z) < 0\}$ as $\mathcal{D}'$. Proving that the affine form $\psi(z)$ is non-negative for all points in $\mathcal{D}$ is equivalent to proving that the new domain $\mathcal{D}'$ is empty. If $\mathcal{D}'$ is *not* empty then, there are some points in $\mathcal{D}$ for which $\psi(z)$ is *not* non-negative.

4.2 Legality of Space-time Mapping for Reductions

We have seen that for a reduction expression, the space-time mapping can be specified either for the reduction body or the reduction result. We will now discuss the legality conditions for each of these cases. For rest of this section, let us consider a reduction equation as follows

$$A[z] = \texttt{reduce}(\oplus,\ p : z' \rightarrow z,\ g(\ldots B[f(z')]\ldots)) \tag{4.1}$$

In this equation, let D_A be the domain of variable A and let $\mathcal{D}_{\text{body}}$ be the context domain of the reduction body. For each point z in D_A, the expression $g(\ldots B[f(z')]\ldots))$ is evaluated for a set of points $\{z'|z' \in \mathcal{D}_{\text{body}}\ ;\ p(z') = z\}$ and reduced by applying the operator $\oplus$ to produce the value at $A[z]$.

In order to make our explanation of the two cases homogeneous, it is convenient to rewrite equation 4.1 as

$$A[z] = \texttt{reduce}(\oplus,\ p : z' \rightarrow z,\ R_A[z']) \tag{4.2}$$

$$R_A[z'] = g(\ldots B[f(z')]\ldots) \tag{4.3}$$

where, the domain of the variable R_A is $\mathcal{D}_{\text{body}}$.

When space-time mapping is specified for reduction body, then it should be treated as the space-time mapping for the variable R_A. If the space-time mapping is specified for reduction result then, it should be treated as the space-time mapping for variable A. In either case, only one of the two variables A or R_A will have the space-time mapping.

4.2.1 Legality of Space-time Mapping for Reduction Body

If the space-time mapping is specified for the reduction body then, it means that the associated function ϕ_A specifies the time and processor at which the expression

$g(\ldots B[f(z')]\ldots))$ is evaluated for each point z' in $\mathcal{D}_{\text{body}}$. In order to evaluate $g(\ldots B[f(z')]\ldots))$ for each point z' in $\mathcal{D}_{\text{body}}$ the requirement is that all the values needed in the expression g should be available. In this case it is at least the value $B[f(z')]$.

For ϕ_{R_A} to be legal, the first requirement is that all the dependences for which the consumer is the variable R_A should be satisfied. In our case, there is at least one dependence namely $\forall z \in \mathcal{D}_{\text{body}} : \langle R_A, z' \rangle \rightarrow \langle B, f(z') \rangle$. So the legality condition for the dependence $\forall z \in \mathcal{D}_{\text{body}} : \langle R_A, z' \rangle \rightarrow \langle B, f(z') \rangle$ to be satisfied is

$$\forall z' \in \mathcal{D}_{\text{body}} : \phi_{R_A}(z') \rhd \phi_B(f(z'))$$

We have till now only solved a part of the problem, the real difficulty is with the dependences where the variable A is a producer. For example, along with the equations defining variables A and R_A, consider another equation as follows

$$C[z''] = g_c(\ldots A[f_c(z'')]\ldots)$$

For the dependence $\forall z'' \in \mathcal{D}_C : \langle C, z'' \rangle \rightarrow \langle A, f_c(z'') \rangle$ to be satisfied

$$\forall z'' \in \mathcal{D}_C : \phi_C(z'') \rhd \phi_A(f_c(z''))$$

Since we do not have ϕ_A, we have no information about when each of the $A[f_c(z'')]$ are produced. But we know when each value in the set $\{z' | z' \in \mathcal{D}_{\text{body}} \; ; \; p(z') = f(z'')\}$ which contribute to the value $A[f_c(z'')]$ are produced.

One way to solve this problem is to deduce ϕ_A. For each point $f_c(z'')$ in the domain of the variable A, we can say $\phi_A(f_c(z''))$ at which $A[f_c(z'')]$ is computed is the $\phi_{R_A}(z'))$ of the point in the set $\{z' | z' \in \mathcal{D}_{\text{body}} \; ; \; p(z') = f(z'')\}$ that has *last contributed* to the value at $A[f_c(z'')]$. We can get this by doing *change of basis* operation on $\mathcal{D}_{\text{body}}$ with function ϕ_{R_A} and then taking *lexicographic maximum* of the resulting polyhedra parametrized by $f(z'')$.

Alternatively, an easier way to check $\phi_C(z'') \rhd \phi_A(f_c(z''))$ is to check $\phi_C(z'') \rhd \phi_{R_A}(z')$ for all points z' in the set $\{z' | z' \in \mathcal{D}_{\text{body}} \ ; \ p(z') = f(z'')\}$ that contribute to the value $A[f_c(z'')]$. So for the dependence $\forall z'' \in \mathcal{D}_C : \langle C, z'' \rangle \rightarrow \langle A, f_c(z'') \rangle$ to be satisfied

$$\forall \langle z'', z' \rangle \in \mathcal{P} \ : \ \phi_C(z'') \rhd \phi_{R_A}(z')$$

where, $\mathcal{P} = \{\langle z'', z' \rangle \mid z'' \in D_C \ ; \ z' \in \mathcal{D}_{\text{body}} \ ; \ f(z'') \in D_A \ ; \ p(z') = f(z'')\}$

4.2.2 Legality of Space-time Mapping for Reduction Result

For the reduction equation, if the space-time mapping is specified for the result of the reduction then, for each point z in D_A, $\phi_A(z)$ gives the iteration vector at which the value at $A[z]$ is produced. With respect to code generation for reduction, the code generator cannot add an extra loop for reduction as the depth of loop nest it generates is strictly equal to the number of `SEQUENTIAL` and `PARALLEL` dimensions in the space-time mapping specification. So the code generator creates a separate function which contains the extra loop(s) required for the reduction.

For each point z in D_A, the reduction function would require all the values necessary to produce the value at $A[z]$. If $\phi_A(z)$ is later than all iteration vectors that have produced the values required for $A[z]$ then the space-time mapping ϕ_A is legal.

Each point z' in $\mathcal{D}_{\text{body}}$ contributes to a point $p(z')$ in D_A. So for each point z' in $\mathcal{D}_{\text{body}}$ the operation $\langle A, p(z') \rangle$ depends on the operation $\langle R_A, z' \rangle$ which in-turn depends on the operation $\langle B, f(z') \rangle$. Indirectly, $\langle A, p(z') \rangle$ depends on $\langle B, f(z') \rangle$. Since we do not have ϕ_{R_A} but rather know ϕ_A, the Verifier checks if the indirect dependence $\forall z' \in \mathcal{D}_{\text{body}} : \langle A, p(z') \rangle \rightarrow \langle B, f(z') \rangle$ is satisfied. The condition the Verifier checks is

$$\forall z' \in \mathcal{D}_{\text{body}} : \phi_A(p(z')) \rhd \phi_B(f(z'))$$

Note that this condition is equivalent to the one specified by the Redon-Feautrier scheduler [41].

4.3 Legality of Memory Mapping Specification

When a variable is given a many-to-one memory mapping then, multiple points in the domain of the variable will write into the same memory location. Having many-to-one memory mapping gives rise to the question "when can a memory location be overwritten". The answer to this is that "overwriting of a memory location should happen only after all the *required reads* of the value that is already in that location [17]".

Consider an Alphabets program with variables A and B defined over domains D_A and D_B respectively and equations of the form

$$B[z'] = \ldots$$
$$A[z] = g(\ldots B[f(z)] \ldots)$$

In the target mapping specification for this program, let ϕ_A and ϕ_B be the space-time mappings for variables A and B respectively. Let variable B be given a many-to-one memory mapping m_B.

In the dependence $\forall z \in \mathcal{D}_A : \langle A, z\rangle \rightarrow \langle B, f(z)\rangle$. Let z_1 be a point in $\mathcal{D}_A$ that reads from a point $f(z_1)$ in D_B. Let z_2' be a point in D_B which *overwrites* the same memory location that $f(z_1)$ refers to. i.e.,

$$m_B(z_2') = m_B(f(z_1)) \text{ and } \phi_B(z_2') \rhd \phi_B(f(z_1))$$

The memory mapping m_B is *illegal* if the overwriting happens *before* the earlier value i.e., $B[f(z_1)]$ is read by the point z_1 in $\mathcal{D}_A$. i.e., if

$$\phi_A(z_1) \rhd \phi_B(z_2')$$

The Verifier checks if the above condition is false for all the points in $\mathcal{D}_A$. The problem formulated by the Verifier is as follows

$$\forall \langle z', z \rangle \in \mathcal{P} \ : \ \phi_B(z') \rhd \phi_A(z) \tag{4.4}$$

where, $\mathcal{P} = \{\langle z', z \rangle \mid z' \in D_B \ ; \ z \in \mathcal{D}_A \ ; \ m_B(z') = m_B(f(z)) \ ; \ \phi_B(z') \rhd \phi_B(f(z))\}$.

The memory mapping m_B is said to be legal, iff the condition in the equation 4.4 is satisfied for all the dependences in which the variable B is the producer.

There is a special case where the *overwriting operation and the reading operation are the same*. This is very common with variables used for serializing reductions. In these cases we relax the legality condition to "overwriting should happen after or at the same time as the read", i.e., we formulate the constraints as non-strict inequalities.

4.3.1 Legality of Memory Space Sharing

The legality condition for memory space sharing is similar to that of legality condition for memory mapping, where overwriting of a memory location should only happen after all the required read of the value that is already in that location. Only difference is that the overwriting operation will now be from an equation of a different variable.

Consider an Alphabets program with variables A, B and C defined over domains D_A, D_B and D_C respectively and equations of the form

$$B[z''] = \ldots$$
$$A[z] = g(\ldots B[f(z)] \ldots)$$
$$C[z'] = \ldots$$

In the target mapping specification for this program, let ϕ_A, ϕ_B and ϕ_C be the space-time mappings for variables A, B and C respectively. Let variables B and

C share the memory space and let $m_B(z'')$ and $m_C(z')$ be their respective memory mapping functions. Let z'' and z' be two points in the domains of the variables B and C respectively. Then,

$$m_B(z'') = m_C(z')$$

In the dependence $\forall z \in \mathcal{D}_A : \langle A, z\rangle \rightarrow \langle B, f(z)\rangle$. Let z_1 be a point in $\mathcal{D}_A$ that reads from a point $f(z_1)$ in D_B. Let z_2'' be a point in D_C which *overwrites* the same memory location that $f(z_1)$ refers to. i.e.,

$$m_C(z_2'') = m_B(f(z_1)) \text{ and } \phi_C(z_2'') \rhd \phi_B(f(z_1))$$

The memory space sharing is *illegal* if the overwriting happens *before* the read by the point z_1 in $\mathcal{D}_A$. i.e., if

$$\phi_A(z_1) \rhd \phi_C(z_2'')$$

The Verifier checks if the above condition is false for all the points in $\mathcal{D}_A$. The problem formulated by the Verifier is as follows

$$\forall \langle z'', z\rangle \in \mathcal{P} \ : \ \phi_C(z'') \rhd \phi_A(z) \tag{4.5}$$

where, $\mathcal{P} = \{\langle z'', z\rangle \mid z'' \in D_C \ ; \ z \in \mathcal{D}_A \ ; \ m_c(z'') = m_B(f(z)) \ ; \ \phi_C(z'') \rhd \phi_B(f(z))\}$.

4.4 Legality of Tiling Specification

Tiling partitions the iteration space of a loop nest into smaller blocks for cache locality or/and coarse-grain parallelism. When a loop nest is tiled, the execution order of operations may no longer be same as that in the original loop nest. Let S_1 and S_2 be two statements in a sequential loop nest. Let z_1 and z_2 be iteration vectors in the domains of S_1 and S_2 respectively and $z_2 \rhd z_1$. So $\langle S_2, z_2\rangle \gg \langle S_1, z_1\rangle$.

When this loop nest is tiled, let z_1' and z_2' be the corresponding iteration vectors for z_1 and z_2. Even though the precedence $z_2' \rhd z_1'$ is preserved in the tiled loop nest, but we cannot say $\langle S_2, z_2' \rangle \gg \langle S_1, z_1' \rangle$. This is only guaranteed if z_1 is in the first orthant of z_2 in the original loop nest, i.e., if z_1 can be expressed as $z_2 + \vec{k}$, where, all the components of $\vec{k}$ are greater than or equal to zero. If a parallel loop nest is tiled then, in order to preserve the precedence relationship between two operations, the corresponding iteration vectors have to satisfy extra constraints.

Because tiling a loop nest changes the execution order of operations, some of the dependences in the original loop nest may be violated in the tiled loop nest. We will now see the sufficient conditions for dependences in the original loop nest to be satisfied if a loop nest is tiled. Consider a loop nest as follows

Listing 4.1: Tiling example 1

```
for(int i = 0; i < N; i++) {
  for(int j = 0; j < i; i++) {
  S1: A[i][j] = A[i-1][j+1]
  }
}
```

In this loop nest, consider an operation $\langle S_1, i_t, j_t \rangle$ that depends on the operation $\langle S_1, i_t - 1, j_t + 1 \rangle$. Since the loop executes in the lexicographic order of iteration vectors, $\langle S_1, i_t, j_t \rangle \gg \langle S_1, i_t - 1, j_t + 1 \rangle$.

If this loop nest is tiled, then the operations $\langle S_1, i_t, j_t \rangle$ and $\langle S_1, i_t - 1, j_t - 1 \rangle$ may be mapped to different tiles, as shown in figure 4.2. If this is the case then, in the execution order of tiles, the tile that contains $[i_t, j_t]$ will execute before the tile that contains $[i_t - 1, j_t + 1]$. So the condition $\langle S_1, i_t, j_t \rangle \gg \langle S_1, i_t - 1, j_t + 1 \rangle$ will no longer be true.

If the loops to be tiled are all sequential then, legality of tiling depends on the permutability of loops. If a set of loops are fully permutable then, tiling of those loops is legal.

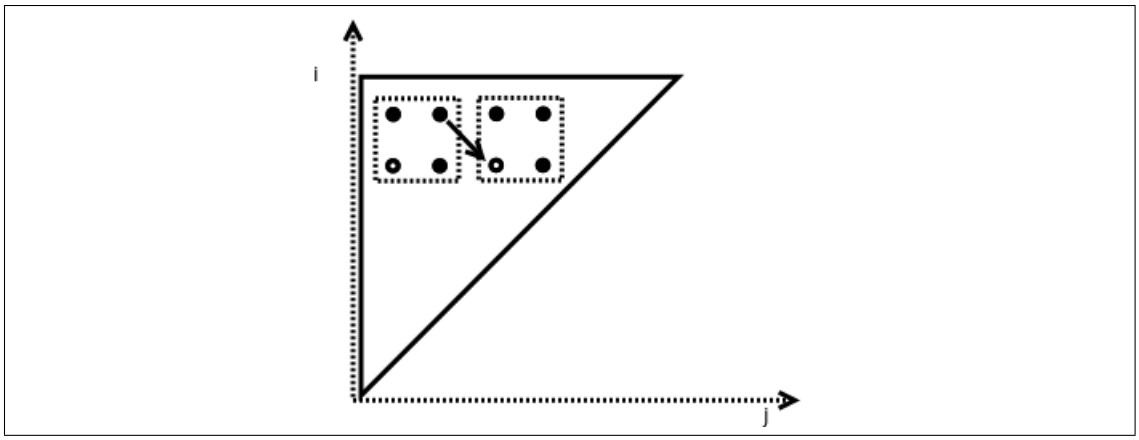

Figure 4.2: Incorrect tiling example

Fully Permutable Loops Consider a hypothetical n dimensional loop nest with statement $S_1, S_2, \ldots S_t$. In this loop nest, let z_1 and z_2 be two iteration vectors in the domains of statements S_1 and S_2 respectively. Additionally assume that the operation $\langle S_2, z_2 \rangle$ depends on $\langle S_1, z_1 \rangle$ and $\langle S_2, z_2 \rangle \gg \langle S_1, z_1 \rangle$.

In this loop nest, let $\{L_i, L_{i+1}, \ldots L_j\}$ be a set of sequential loops which we wish to tile. The condition $\langle S_2, z_2 \rangle \gg \langle S_1, z_1 \rangle$ will be true in the tiled loop nest if either of the following conditions are satisfied

- $z_2 \rhd_{i-1} z_1$. i.e., $z_2 \rhd z_1$ is satisfied in the first $i-1$ components of the two vectors.

- If $z_2 =_{i-1} z_1$, i.e., if the first $i-1$ components of the two vector are equal then, $\forall k \ : \ z_2^k \geq z_1^k$. where, z_2^k and z_1^k are the components of vectors z_2 and z_1 in the k^{th} dimension and $\{i \leq k \leq j\}$.

Iff all the dependences in the loop nest satisfy the above condition then, the loops $\{L_i, L_{i+1}, \ldots L_j\}$ are said to be fully permutable.

Legality of Tiling Parallel Loops If a parallel loop is to be tiled then, for each dependence, the corresponding components of the iteration vectors of producer and consumer of the dependence should be same. For example, If the loop nest in listing 4.2 is tiled, since the loop j is parallel, all the tiles in a row will execute in parallel. Because of this there will be dependences between operations in tiles that execute in parallel.

Listing 4.2: Tiling example 2

```
for(int i = 0; i < N; i++) {
  #pragma omp parallel for
  for(int j = 0; j < i; j++) {
    A[i][j] = A[i-1][j-1]
  }
}
```

The figure 4.3 shows two tiles that execute in parallel and the dependence crossing the tile boundary. Tiling of a parallel loop is legal *iff* operations in a tile depend on operations in the same tile or on the operations that are mapped to the tiles that have already executed.

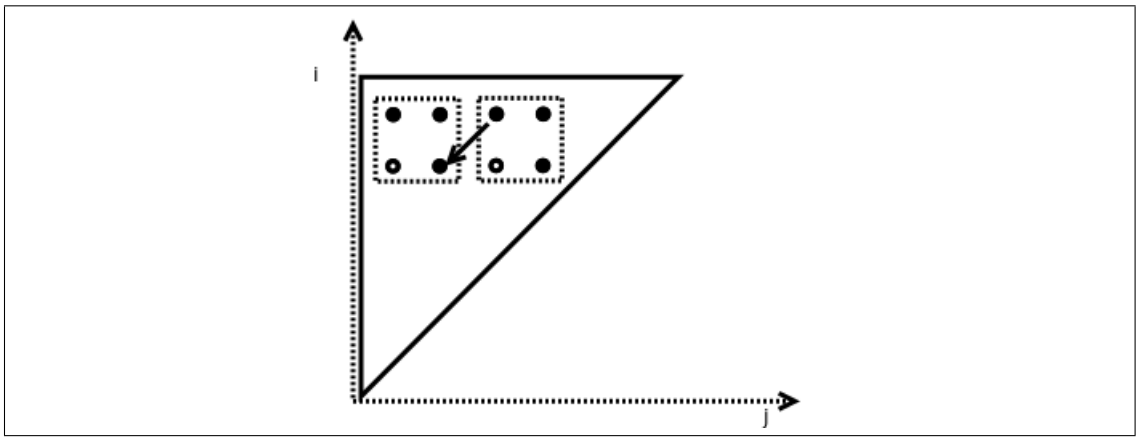

Figure 4.3: Incorrect tiling example2

Legality of Tiling Specification In the tiling part of the target mapping specification, the user specifies which dimensions in the space-time mapping should be tiled.

The Verifier checks the legality of tiling specification based on the rules that we have defined in *fully permutable loops* and *legality of tiling parallel loops*.

In an Alphabets program, let n denote a variable defined over a domain D_n. Let V denote the set of all the variables in the program. Let e denote a dependence and $C(e)$ be the consumer of the dependence and $P(e)$ be the producer. let f_e denote the dependence function and $\mathcal{D}_e$ be the context domain of the dependence. Let E denote the set of all the dependences in the program.

Given a space-time mapping specification Φ for the program, where ϕ_n denotes space-time mapping for variable n. And if $\{i, i+1, \ldots j\}$ are the dimensions that user wants the code generator to tile.

The Verifier checks $\forall e \in E$ and which are *not* satisfied in the dimensions prior to i. $\forall k : i \leq k \leq j$

- if k is annotated as `SEQUENTIAL` then $\forall z \in \mathcal{D}_e : \phi^k_{c(e)}(z) \geq \phi^k_{p(e)}(f_e(z))$
- if k is annotated as `PARALLEL` then $\forall z \in \mathcal{D}_e : \phi^k_{c(e)}(z) = \phi^k_{p(e)}(f_e(z))$

Chapter 5

Conclusions, Limitations & Future Work

5.1 Limitations & Future Work

Mod-Factors In the AlphaZ system, While specifying a memory mapping for a variable, users can also specify *mod-Factor* for the memory mapping, currently the verifier does not check the legality of such memory mapping. In other words, the mod-factor is assumed to be 1 by the verifier.

Synchronization Currently the verifier is tightly coupled with scheduled-C code generator. Since scheduled-C generates OpenMP parallel code, the legality conditions that the verifier checks are based on semantics of OpenMP. Precisely, semantics of OpenMP requires that each iteration of a parallel loop be independent, so the verifier strictly adheres to this, and does not allow any dependences to cross processor boundaries (unless they are already satisfied in outer dimensions). In future if AlphaZ incorporates other code generators which can produce code in which iterations of parallel loops can be synchronized (by placing barriers etc), then the verifier will have to be extended.

5.2 Conclusions

Nevertheless, the verifier in its current state has been used as one of the pedagogic tools in teaching a graduate level course which introduces programming in the polyhedral model. Also the same verifier is used as a backend in the tool ompVerify [33] an eclipse plug-in that can detect semantic bugs caused due to incorrect parallelization of loops in affine control parts of OpenMP programs.

REFERENCES

[1] P. Feautrier, "Dataflow analysis of array and scalar references," *International Journal of Parallel Programming*, vol. 20, no. 1, pp. 23–53, 1991.

[2] C. Bastoul, "Code generation in the polyhedral model is easier than you think," in *PACT'13: IEEE International Conference on Parallel Architectures and Compilation and Techniques*, Juan-les-Pins, September 2004, pp. 7–16.

[3] S. Pop, A. Cohen, C. Bastoul, S. Girbal, P. Jouvelot, G.-A. Silber, and N. Vasilache, "GRAPHITE: Loop optimizations based on the polyhedral model for GCC," in *Proc. of the 4th GCC Developper's Summit*, Ottawa, Canada, Jun. 2006, pp. 179–198.

[4] T. Grosser, H. Zheng, R. A, A. Simbürger, A. Grösslinger, and L.-N. Pouchet, "Polly - polyhedral optimization in LLVM," in *First International Workshop on Polyhedral Compilation Techniques (IMPACT'11)*, Chamonix, France, Apr. 2011.

[5] "IBM XL Compilers." [Online]. Available: http://www-01.ibm.com/software/awdtools/xlcpp/

[6] R. Lethin, A. Leung, B. Meister, and E. Schweitz, "R-Stream R-Stream: A parametric high level compiler."

[7] S. V. Rajopadhye, S. Purushothaman, and R. M. Fujimoto, "On synthesizing systolic arrays from recurrence equations with linear dependencies," in *Proceedings, Sixth Conference on Foundations of Software Technology and Theoretical Computer Science*. New Delhi, India: Springer Verlag, LNCS 241, December 1986, pp. 488–503, later appeared in Parallel Computing, June 1990.

[8] P. Feautrier, "Some efficient solutions to the affine scheduling problem. I. One-dimensional time," *International Journal of Parallel Programming*, vol. 21, no. 5, pp. 313–347, 1992.

[9] ——, "Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time," *International Journal of Parallel Programming*, vol. 21, no. 6, pp. 389–420, 1992.

[10] W. Pugh and D. Wonnacott, "Eliminating false data dependences using the Omega test," in *Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, ser. PLDI '92. New York, NY, USA: ACM, 1992, pp. 140–151. [Online]. Available: http://doi.acm.org/10.1145/143095.143129

[11] ——, "Constraint-based array dependence analysis," *ACM Trans. Program. Lang. Syst.*, vol. 20, pp. 635–678, May 1998. [Online]. Available: http://doi.acm.org/10.1145/291889.291900

[12] F. Quilleré, S. Rajopadhye, and D. Wilde, "Generation of efficient nested loops from polyhedra," *International Journal of Parallel Programming*, vol. 28, no. 5, pp. 469–498, October 2000.

[13] G. Gupta and S. V. Rajopadhye, "Simplifying reductions," in *POPL'06*, 2006, pp. 30–41.

[14] P. Srinivasa, "Code generation in AlphaZ," Master's thesis, Colorado State University, 2010.

[15] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick, "A view of the parallel computing landscape," *Commun. ACM*, vol. 52, pp. 56–67, October 2009. [Online]. Available: http://doi.acm.org/10.1145/1562764.1562783

[16] W. Pugh and D. Wonnacott, "An exact method for analysis of value-based array data dependences," in *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*. London, UK: Springer-Verlag, 1994, pp. 546–566. [Online]. Available: http://dl.acm.org/citation.cfm?id=645671.665398

[17] F. Quilleré and S. V. Rajopadhye, "Optimizing memory usage in the polyhedral model," *ACM Trans. Program. Lang. Syst.*, vol. 22, no. 5, pp. 773–815, 2000.

[18] C. Chen, "Model-guided empirical optimization for memory hierarchy," Ph.D. dissertation, University of Southern California, May 2007. [Online]. Available: http://www.cs.utah.edu/~chunchen/papers/thesis_chen.pdf

[19] C. Bastoul, A. Cohen, S. Girbal, S. Sharma, and O. Temam, "Putting polyhedral loop transformations to work," in *Workshop on Languages and Compilers for Parallel Computing (LCPC'03)*, ser. LNCS. College Station, Texas: Springer-Verlag, Oct. 2003, pp. 23–30.

[20] C. Bastoul and P. Feautrier, "More legal transformations for locality," in *Euro-Par'10 International Euro-Par conference, LNCS 3149*. Pisa, Italy: Springer-Verlag, Aug. 2004, pp. 272–283, classement CORE : A, nombre de papiers acceptés : 124, soumis : 352, distinguished paper award avec 3 autres papiers. [Online]. Available: http://hal.inria.fr/inria-00001056

[21] M. W. Hall, J. Chame, C. Chen, J. Shin, G. Rudy, and M. M. Khan, "Transformation recipes for code generation and auto-tuning," in *Proceedings of the 22nd International Workshop on Languages and Compilers for Parallel Computing*, Oct 2009.

[22] W. Pugh and E. Rosser, "Iteration space slicing for locality," in *Proceedings of the 12th International Workshop on Languages and Compilers for Parallel Computing*, ser. LCPC '99. London, UK: Springer-Verlag, 2000, pp. 164–184. [Online]. Available: http://dl.acm.org/citation.cfm?id=645677.663787

[23] L.-N. Pouchet, U. Bondhugula, C. Bastoul, A. Cohen, J. Ramanujam, P. Sadayappan, and N. Vasilache, "Loop transformations: convexity, pruning and optimization," in *POPL*, 2011, pp. 549–562.

[24] D. Kim and S. V. Rajopadhye, "Efficient tiled loop generation: D-Tiling," in *LCPC*, 2009, pp. 293–307.

[25] L.-N. Pouchet, "Iterative optimization in the polyhedral model," Ph.D. dissertation, University of Paris-Sud 11, Orsay, France, Jan. 2010.

[26] D. Wonnacott, "Time skewing for parallel computers," in *Proceedings of the 12th International Workshop on Languages and Compilers for Parallel Computing*, ser. LCPC '99. London, UK: Springer-Verlag, 2000, pp. 477–480. [Online]. Available: http://dl.acm.org/citation.cfm?id=645677.663799

[27] U. Bondhugula, M. M. Baskaran, A. Hartono, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan, "Towards effective automatic parallelization for multicore systems," in *IPDPS*, 2008, pp. 1–5.

[28] N. Vasilache, C. Bastoul, and A. Cohen, "Polyhedral code generation in the real world," in *Proceedings of the International Conference on Compiler Construction (ETAPS CC'06)*, ser. LNCS 3923. Vienna, Austria: Springer-Verlag, Mar. 2006, pp. 185–201, classement CORE : A, nombre de papiers acceptés : 20, soumis : 71.

[29] C. Ancourt and F. Irigoin, "Scanning polyhedra with DO loops," 1991, pp. 39–50.

[30] C. Bastoul and P. Feautrier, "Adjusting a program transformation for legality," *Parallel processing letters*, vol. 15, no. 1, pp. 3–17, Mar. 2005, classement CORE : U.

[31] N. Vasilache, C. Bastoul, A. Cohen, and S. Girbal, "Violated dependence analysis," in *Proceedings of the 20th annual international conference on Supercomputing*, ser. ICS '06. New York, NY, USA: ACM, 2006, pp. 335–344. [Online]. Available: http://doi.acm.org/10.1145/1183401.1183448

[32] N. Vasilache, A. Cohen, and L.-N. Pouchet, "Automatic correction of loop transformations," in *16th International Conference on Parallel Architecture and Compilation Techniques (PACT'07)*. Brasov, Romania: IEEE Computer Society Press, September 2007, pp. 292–304.

[33] V. Basupalli, T. Yuki, S. V. Rajopadhye, A. Morvan, S. Derrien, P. Quinton, and D. Wonnacott, "ompVerify: Polyhedral analysis for the OpenMP programmer," in *IWOMP*, 2011, pp. 37–53.

[34] C. Bastoul, "Improving data locality in static control programs," Ph.D. dissertation, University Paris 6, Pierre et Marie Curie, France, Dec. 2004.

[35] U. K. R. Bondhugula, "Effective automatic parallelization and locality optimization using the polyhedral model," Ph.D. dissertation, Columbus, OH, USA, 2008, adviser-Sadayappan, P.

[36] G. Gupta, "Some advances in the polyhedral model," CSU Theses/Dissertation, 2010. [Online]. Available: http://hdl.handle.net/10217/40288

[37] D. K. Wilde, "The ALPHA language," 38330 montbonnot st martin unite de recherche inria rocquencourt, domaine de voluceau, rocquencourt, bp 105, 78153 le chesnay cedex unite de recherche inria sophia-antipolis, 2004 route des lucioles, bp 93, 06902 sophia-antipolis cedex editeur inria, doma, Tech. Rep., 1994.

[38] G. Gupta and S. Rajopadhye, "The Alphabets langauge specification." [Online]. Available: http://www.cs.colostate.edu/~cs560/Lectures/

[39] W. A. Kelly, A. Professor, and W. W. Pugh, "Optimization within a unified transformation framework," 1996.

[40] A. Cohen, S. Girbal, and O. Temam, "A polyhedral approach to ease the composition of program transformations," in *in: Euro-Par'04, no. 3149 in LNCS*. Springer-Verlag, 2004, pp. 292–303.

[41] X. Redon and P. Feautrier, "Scheduling reductions," in *Proceedings of the 8th international conference on Supercomputing*, ser. ICS '94. New York, NY, USA: ACM, 1994, pp. 117–125. [Online]. Available: http://doi.acm.org/10.1145/181181.181319

[42] M.-W. Benabderrahmane, L.-N. Pouchet, A. Cohen, and C. Bastoul, "The polyhedral model is more widely applicable than you think," in *Proceedings of the International Conference on Compiler Construction (ETAPS CC'10)*, ser. LNCS. Paphos, Cyprus: Springer-Verlag, Mar. 2010, classement CORE : A, nombre de papiers acceptés : 16, soumis : 56.

[43] C. Bastoul, "Efficient code generation for automatic parallelization and optimization," in *ISPDC'2 IEEE International Symposium on Parallel and Distributed Computing*, Ljubljana, Slovenia, Oct. 2003, pp. 23–30.

[44] C. Bastoul and P. Feautrier, "Improving data locality by chunking," in *CC'12 International Conference on Compiler Construction, LNCS 2622*, Warsaw, Poland, Apr. 2003, pp. 320–335. [Online]. Available: http://hal.inria.fr/inria-00001055

[45] S. Verdoolaege, "isl: an integer set library for the polyhedral model," in *Proceedings of the Third international congress conference on Mathematical software*, ser. ICMS'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 299–302. [Online]. Available: http://dl.acm.org/citation.cfm?id=1888390.1888455

[46] F. Remondino and N. Bőrlin, "Polylib - a library of polyhedral functions. http://icps.u-strasbg.fr/polylib/ or http://www.irisa.fr/polylib," in *International Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences, Vol. XXXIV, Part 5/W16, H.-G. Maas and D. Schneider (Eds*, 2004.

[47] P. Feautrier, "Parametric integer programming," *RAIRO Recherche Opérationnelle*, vol. 22, no. 3, pp. 243–268, 1988.

[48] J. Xue, "On tiling as a loop transformation," *Parallel Processing Letters*, vol. 7, no. 4, pp. 409–424, 1997.

[49] C. Chen, J. Chame, and M. W. Hall, "CHiLL: A framework for composing high-level loop transformations," University of Southern California, Technical Report 08-897, Jun 2008. [Online]. Available: http://www.cs.usc.edu/research/08-897.pdf

[50] W. Kelly and W. Pugh, "A framework for unifying reordering transformations," College Park, MD, USA, Tech. Rep., 1993.

[51] D. Wilde and S. Rajopadhye, "Memory reuse analysis in the polyhedral model," in *Parallel Processing Letters*. Springer-Verlag, 1996, pp. 389–397.

[52] W. Pugh, "The Omega test: a fast and practical integer programming algorithm for dependence analysis," in *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*. ACM, 1991, p. 13.

[53] J.-F. Collard, *Reasoning about program transformations - imperative programming and flow of data*. Springer, 2003.

[54] R. M. Karp, R. E. Miller, and S. Winograd, "The organization of computations for uniform recurrence equations," *J. ACM*, vol. 14, pp. 563–590, July 1967. [Online]. Available: http://doi.acm.org/10.1145/321406.321418

[55] L. Lamport, "The parallel execution of DO loops," *Commun. ACM*, vol. 17, pp. 83–93, February 1974. [Online]. Available: http://0-doi.acm.org.millennium.lib.cyut.edu.tw/10.1145/360827.360844

[56] R. Seater and D. Wonnacott, "Polynomial time array dataflow analysis," in *Proceedings of the 14th international conference on Languages and compilers for parallel computing*, ser. LCPC'01. Berlin, Heidelberg: Springer-Verlag, 2003, pp. 411–426. [Online]. Available: http://dl.acm.org/citation.cfm?id=1769331.1769358

[57] D. Wonnacott, "Extending scalar optimizations for arrays," in *Proceedings of the 13th International Workshop on Languages and Compilers for Parallel Computing-Revised Papers*, ser. LCPC '00. London, UK: Springer-Verlag, 2001, pp. 97–111. [Online]. Available: http://dl.acm.org/citation.cfm?id=645678.663949

[58] L.-N. Pouchet, U. Bondhugula, C. Bastoul, A. Cohen, J. Ramanujam, and P. Sadayappan, "Combined iterative and model-driven optimization in an automatic parallelization framework," in *SC*, 2010, pp. 1–11.

[59] S. Tavarageri, A. Hartono, M. Baskaran, L.-N. Pouchet, J. Ramanujam, and P. Sadayappan, "Parametric tiling of affine loop nests," in *15th Workshop on Compilers for Parallel Computing (CPC'10)*, Vienna, Austria, Jul. 2010.

[60] L.-N. Pouchet, C. Bastoul, and A. Cohen, "LetSee: the LEgal Transformation SpacE Explorator," Third International Summer School on Advanced Computer Architecture and Compilation for Embedded Systems (ACACES'07), L'Aquila, Italia, July 2007, extended abstract, pp 247–251.

[61] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, and O. Temam, "Semi-automatic composition of loop transformations for deep

parallelism and memory hierarchies," *International Journal of Parallel Programming*, vol. 34, no. 3, pp. 261–317, Jun. 2006, classement CORE : A.

[62] M. E. Wolf and M. S. Lam, "A loop transformation theory and an algorithm to maximize parallelism," *IEEE Trans. Parallel Distrib. Syst.*, vol. 2, pp. 452–471, October 1991. [Online]. Available: http://dx.doi.org/10.1109/71.97902

[63] G. Gupta, S. V. Rajopadhye, and P. Quinton, "Scheduling reductions on realistic machines," in *SPAA*, 2002, pp. 117–126.