DISSERTATION

OPTIMIZATIONS OF POLYHEDRAL REDUCTIONS AND THEIR USE IN ALGORITHM-BASED FAULT TOLERANCE

Submitted by

Louis Narmour

Department of Computer Science

In partial fulfillment of the requirements

For the Degree of Doctor of Philosophy

Colorado State University

Fort Collins, Colorado

Spring 2025

Doctoral Committee:

Advisor: Sanjay Rajopadhye

Louis-Noël Pouchet
Vinayak Prabhu
Ali Pezeshki

ABSTRACT

OPTIMIZATIONS OF POLYHEDRAL REDUCTIONS AND THEIR USE IN ALGORITHM-BASED FAULT TOLERANCE

In this dissertation, we study the optimization of programs containing reductions and motivate a deeper connection between two ostensibly unrelated problems, one involving techniques for algorithmic improvement and another in the domain of Algorithm-Based Fault Tolerance. Reductions combine collections of inputs with an associative and often commutative operator to produce collections of outputs. Such operations are interesting because they often require special handling to obtain good performance. When the same value contributes to multiple outputs, there is an opportunity to reuse partial results, enabling reduction simplification. Prior work showed how to exploit this and obtain a reduction (pun intended) in the program's asymptotic complexity through a program transformation called simplification. We propose extensions to prior work on simplification and provide the first complete push-button implementation of reduction simplification in a compiler and show how to handle a strictly more general class of programs than previously supported. We evaluate its effectiveness and show that simplification rediscovers several key results in algorithmic improvement across multiple domains, previously only obtained through clever manual human analysis and effort. Additionally, we complement this and study the automation of generalized and automated fault tolerance against transient errors, such as those occurring due to cosmic radiation or hardware component aging and degradation, using Algorithm-Based Fault Tolerance (ABFT). ABFT methods typically work by adding some redundant computation in the form of invariant checksums (i.e., reductions), which, by definition, should not change as the program executes. By computing and monitoring checksums, it is possible to detect errors by observing differences in the checksum values. However, this is challenging for two key reasons: (1) it requires careful manual analysis of the input program, and (2)

care must be taken to subsequently carry out the checksum computations efficiently enough for it to be worth it. We propose automation techniques for a class of scientific codes called stencil computations and give methods to carry out this analysis at compile time. This is the first work to propose such an analysis in a compiler.

TABLE OF CONTENTS

Chapter 1

Résumé en français

Cette thèse de doctorat a été réalisée en cotutelle entre l'Université de Rennes, France, et Colorado State University. À ce titre, nous incluons un résumé en français présentant un aperçu de ce travail.

La technologie informatique est devenue de plus en plus puissante et complexe au fil des ans, offrant plus de capacités à chaque nouvelle génération de processeurs. La dernière génération de processeurs Intel Xeon, par exemple, prend en charge des configurations allant jusqu'à 50 cœurs sur une seule puce avec 100 Mo de cache de dernier niveau. Cependant, utiliser toute la puissance de traitement disponible en présence de dépendances de données complexes n'est pas toujours facile. S'appuyer sur des compilateurs traditionnels pour produire un code performant pour un programme d'entrée donné est insuffisant. Une des raisons est que les compilateurs de langages à usage général doivent être très prudents quant aux types d'optimisations qu'ils peuvent employer. En conséquence, il incombe au développeur de l'application d'écrire le programme de manière à ce que le compilateur puisse détecter avec succès les opportunités d'optimisation et générer ensuite un code optimisé. C'est un défi car il est souvent plus facile pour nous de penser aux problèmes à un niveau d'abstraction plus élevé, alors que l'écriture de code efficace nécessite un raisonnement de niveau inférieur. Au fil des ans, cela a conduit au développement d'une grande variété de Langages Spécifiques de Domaine (DSL) et de cadres hautement spécialisés.

Le modèle polyédrique est l'un de ces cadres ; il s'agit d'un formalisme mathématique pour spécifier, analyser et transformer des programmes intensifs en calcul et en données. De tels programmes se retrouvent dans une grande variété de domaines d'application, tels que l'algèbre linéaire dense, le traitement du signal et de l'image, les réseaux de neurones convolutifs, l'apprentissage profond, l'entraînement par rétropropagation, et la programmation

dynamique, pour n'en nommer que quelques-uns. Dans cette thèse, nous abordons deux problèmes apparemment non liés qui s'inscrivent parfaitement dans le cadre de la compilation polyédrique. Le premier problème concerne l'amélioration algorithmique automatique des programmes contenant des réductions, ou scans, à l'aide d'une transformation de programme appelée "simplification". Le second problème concerne l'application automatique de la Tolérance aux Pannes Basée sur les Algorithmes (ABFT). Nous cherchons à motiver la connexion entre ces deux travaux apparemment disjoints.

Les réductions sont omniprésentes en informatique et impliquent généralement l'application d'un opérateur associatif, souvent commutatif, à des collections d'entrées pour produire des collections de résultats. Ces opérations sont intéressantes car elles nécessitent souvent un traitement spécial pour obtenir de bonnes performances. L'API de multithreading OpenMP C/C++ [88] comporte même des directives spécialement conçues pour la parallélisation des réductions. Les optimisations typiques du compilateur offrent au mieux un gain de performance constant. Cependant, dans certains cas, la spécification du programme d'entrée peut impliquer une *réutilisation* — la même valeur contribuant à plusieurs résultats. Lorsqu'elle est correctement exploitée, il est possible d'améliorer la complexité asymptotique de tels programmes. Une amélioration de complexité, de l'ordre de $O(N^3)$ à $O(N^2)$ par exemple, offre une accélération illimitée puisque, asymptotiquement, N peut être arbitrairement grand. Les optimisations de ce type ont traditionnellement reposé sur une analyse humaine très astucieuse et un effort d'ingénierie manuel. Gautam et Rajopadhye [49] ont montré précédemment comment réduire, par de tels degrés polynomiaux, la complexité asymptotique des réductions polyédriques. Ils ont développé une transformation de programme appelée *simplification* et ont esquissé, sans fournir tous les détails nécessaires, un algorithme récursif pour simplifier automatiquement les réductions polyédriques. Cependant, leur travail était principalement théorique et plusieurs problèmes clés devaient être résolus avant de pouvoir écrire un outil pratique. De plus, leur travail n'était pas maximal dans le sens où il ne pouvait pas traiter tous les types de réductions d'entrée possibles. Cette thèse complète, étend

et implémente la transformation de simplification dans une chaîne d'outils de compilation polyédrique et nous étudions son efficacité et son applicabilité aux problèmes du monde réel.

En complément de la performance, nous étudions des techniques pour garantir la correction des programmes. Les erreurs silencieuses transitoires provenant du matériel se manifestent sous la forme de corruptions silencieuses de données et posent un sérieux problème pour la fiabilité des logiciels. Ces erreurs surviennent en raison de phénomènes tels que le rayonnement cosmique [67, 84] et le vieillissement et la dégradation des composants matériels [85, 55]. Il s'agit d'un problème de taille et d'échelle. À une extrémité du spectre, alors que les tendances poussent au développement de systèmes plus petits et à plus faible consommation, la probabilité de rencontrer de telles erreurs augmente [60, 68]. À l'autre extrémité, des erreurs de corruption silencieuse de données ont même été observées dans des services d'infrastructure à grande échelle fonctionnant à l'échelle mondiale aussi récemment qu'en 2021 [87]. Cela a également des implications dans le domaine du calcul haute performance (HPC) [76, 72] où les charges de travail peuvent s'exécuter pendant des semaines, voire des mois. Attendre des mois pour que le calcul se termine seulement pour réaliser que le résultat est incorrect a un impact significatif, compte tenu du temps perdu. Le besoin d'une tolérance aux pannes robuste devient de plus en plus prévalent à mesure que les plateformes informatiques se développent en capacité et en complexité. Une approche pour permettre la tolérance aux pannes implique la duplication, soit dans le logiciel [78, 48], soit dans le matériel [52]. La duplication offre la couverture la plus élevée mais également le coût le plus élevé. D'autres approches évitent la duplication et utilisent l'analyse à la compilation pour augmenter le logiciel avec des sommes de contrôle pour détecter les erreurs dans la mémoire [73]. Bien que cela soit moins coûteux, cela offre également une couverture inférieure. Les erreurs silencieuses qui se produisent ailleurs, à l'intérieur des unités arithmétiques en virgule flottante, passent inaperçues. La détection des erreurs silencieuses (par exemple, transitoires) est difficile car les erreurs se manifestent *dans les données*, et cela nécessite de faire une analyse *sur les données calculées*, ce que la plupart des schémas de tolérance aux pannes

ignorent.

La tolérance aux pannes basée sur les algorithmes (ABFT) [7] a été largement étudiée depuis qu'elle a été proposée pour la première fois en 1984 et, contrairement à la plupart des autres schémas, elle offre un moyen relativement bon marché de détecter et corriger de telles erreurs *dans les données*. L'idée principale est d'augmenter le calcul avec un travail supplémentaire en utilisant des sommes de contrôle invariantes en exploitant des identités algébriques, qui devraient rester de valeur constante pendant l'exécution du programme. En comparant les sommes de contrôle évaluées périodiquement pendant l'exécution du programme avec leurs valeurs calculées séparément de manière équivalente mais simple, nous pouvons détecter des erreurs si leur différence dépasse un certain seuil. Cependant, le principal défi de la tolérance aux pannes basée sur les algorithmes est qu'il n'existe pas de solution universelle. C'est parce que l'ABFT exploite les identités algébriques particulières du programme d'entrée. Historiquement, cela a conduit à un travail complètement distinct pour chaque "nouvelle" application de l'ABFT : multiplication de matrices en ligne [69], factorisation LU [80], décomposition de Cholesky [74], méthodes itératives générales [75], réseaux de transformée de Fourier rapide [32], réseaux de neurones convolutifs [90], et calculs de stencil [82]. Ceci est dû au fait que pour un programme d'entrée arbitraire, effectuer l'ABFT nécessite une analyse manuelle très minutieuse qui ne suit pas le même schéma d'un programme à l'autre. Premièrement, les calculs de sommes de contrôle doivent être construits en fonction des dépendances du programme. Deuxièmement, les calculs de sommes de contrôle doivent être effectués de manière à ne pas dominer le temps d'exécution du programme. Idéalement, la complexité asymptotique des calculs de sommes de contrôle peut être rendue inférieure à la complexité du calcul principal. Avant notre travail, il n'y avait aucun effort pour généraliser l'ABFT malgré son efficacité à détecter des erreurs dans le *calcul* que seuls les schémas de tolérance aux pannes autres que la duplication peuvent détecter de manière fiable. Comme nous le montrerons, la tâche de construire et d'exécuter les sommes de contrôle ABFT convient parfaitement aux analyses polyédriques et plus particulièrement à l'analyse

en ce qui concerne la simplification. C'est parce que la somme de contrôle ABFT peut être considérée comme des instances de réductions qui peuvent être simplifiées. Ce faisant, nous menons une analyse approfondie des programmes de stencil et fournissons une comparaison détaillée de trois considérations de conception différentes. Nous motivons ensuite la connexion entre l'ABFT et la simplification.

Les contributions de cette thèse peuvent être résumées comme suit : Nous fournissons une implémentation open-source de la construction de la treillis de faces pour des polyèdres convexes arbitraires en utilisant la bibliothèque d'ensembles entiers, "isl" [62]. Nous fournissons le premier moteur de simplification entièrement automatique et prêt à l'emploi en tant qu'outil autonome. Nous étendons la transformation de simplification pour prendre en charge une classe de programme strictement plus générale que celle précédemment prise en charge. Nous fournissons des techniques d'automatisation pour effectuer des analyses ABFT dans un compilateur pour une classe de programmes appelés calculs de stencil. Nous fournissons une analyse approfondie des compromis de décision de conception entre les schémas ABFT dans les calculs de stencil. Nous motivons la connexion plus profonde entre l'ABFT et la transformation de simplification dans l'effort pour enlever la charge de l'analyse de l'humain et tirer parti de la puissance des techniques de compilation polyédrique.

Chapter 2

Introduction

Computing technology has become increasingly powerful and complex over the years, offering more capability with each new generation of processors. The latest generation of Intel Xeon processors, for example, supports configurations up to 50 cores on a single die with 100 MB of last-level cache. However, using all the available processing power in the presence of complex data dependencies is not always easy. Relying on traditional compilers to produce high-performance code for a given input program is insufficient. One reason is that general-purpose language compilers must be very conservative in the types of optimizations that can be employed. Consequently, the onus is on the application developer to write the program in such a way that the compiler can successfully detect optimization opportunities and subsequently emit optimized code. This is challenging because it is often easier for us to think about problems from a higher level of abstraction, whereas writing efficient code requires lower-level reasoning. Over the years, this has led to the development of a wide variety of Domain Specific Languages (DSLs) and highly specialized frameworks.

The polyhedral model is one such framework; it is a mathematical formalism for specifying, analyzing and transforming compute-intensive and data-intensive programs. Such programs occur in a wide variety of application domains, like dense linear algebra, signal and image processing, convolutional neural nets, deep learning, back-propagation training, and dynamic programming, to name just a few. In this thesis, we tackle two ostensibly unrelated problems that fit nicely within the polyhedral compilation framework. The first problem relates to automatic algorithmic improvment of programs containing reductions, or scans, using a program transformation called "simplification". The second problem relates to the automatic application of Algorithm-Based Fault Tolerance (ABFT). We seek to motivate the connection between the complementarity of these two bodies of work.

Reductions are ubiquitous in computing and typically involve applying an associative, often commutative, operator to collections of inputs to produce collections of results. Such operations are interesting because they often require special handling to obtain good performance. The OpenMP C/C++ multithreading API [88] even has directives tailored specifically for parallelizing reductions. Typical compiler optimizations yield, at best, a constant fold speedup. However, in some cases, the input program specification may involve *reuse*—the same value contributing to multiple results. When properly exploited, it is possible to improve the asymptotic complexity of such programs. A complexity improvement from, say, $O(N^3)$ to $O(N^2)$ yields unbounded speedup since, asymptotically, N can be arbitrarily large. Optimizations of this type have traditionally relied on very clever and manual human analysis and engineering effort. Gautam and Rajopadhye [49] previously showed how to reduce, by such polynomial degrees, the asymptotic complexity of polyhedral reductions. They developed a program transformation called *simplification* and outlined, without providing all necessary details, a recursive algorithm to automatically simplify polyhedral reductions. However, their work was primarily theoretical and several key problems needed to be solved before one could write a practical tool. Additionally, their work was not maximal in the sense that it could not handle all possible types of input reductions. This thesis augments, extends and implements the simplification transformation in a polyhedral compilation toolchain and we study its effictiveness and applicability to real world problems.

Complementary to performance, we study techniques to ensure program correctness. Transient silent errors originating in the hardware manifest as silent data corruption and pose a serious concern for software reliability. Such errors occur due to phenomena such as cosmic radiation [67, 84] and hardware component aging and degradation [85, 55]. This is a problem of both size and scale. At one end of the spectrum, as trends push the development of smaller and lower power systems, the likelihood of encountering such errors increases [60, 68]. On the other end, silent data corruption errors have even been observed in large-scale infrastructure services running at the global scale as recently as 2021 [87]. This also has

implications in the realm of High-Performance Computing (HPC) [76, 72] where workloads can run for weeks or even months. Waiting months for the computation to finish only to realize that the result is incorrect has a significant impact, given the time wasted. The need for robust fault tolerance is becoming increasingly prevalent as computing platforms grow in capability and complexity. One approach to enable fault tolerance involves duplication, also know as triple-modular redundancy (TRM), either in the software [78, 48] or in the hardware [52]. Duplication has the highest coverage but also the highest overhead. Other approaches avoid duplication and employ compile-time analysis to augment the software with checksums to detect errors in the memory [73]. While this is less expensive, it also has lower coverage. Silent errors that happen elsewhere, inside the floating-point arithmetic units, go undetected. Detection of silent (e.g., transient) errors is difficult because the errors manifest *in the data*, and doing so requires running some analysis *on the computed data*, which most fault tolerance schemes ignore.

Algorithm-based fault tolerance (ABFT) [7] has been widely studied since it was first proposed in 1984 and, unlike most other schemes, provides a relatively cheap way to detect and correct, such errors *in the data*. The main idea is to augment the computation with extra work using invariant checksums by exploiting algebraic identities, which should remain constant-valued as the program executes. By comparing checksums evaluated periodically after the program executes with their values computed separately in equivalent but simple ways, we can detect errors if their difference is above some threshold. However, the main challenge with algorithm-based fault tolerance is that there is no one-size-fits-all. This is because ABFT exploits the particular algebraic identities of the input program. Historically, this has led to a completely separate work for each "new" application of ABFT: online matrix multiplication [69], LU factorization [80], Cholesky decomposition [74], general iterative methods [75], Fast-Fourier Transform networks [32], convolutional neural networks [90], and stencil computations [82]. This is because for an arbitrary input program, performing ABFT requires very careful manual analysis that does not fit the same pattern across programs.

First, the checksum computations must be constructed as a function of the program dependencies. Second, the checksum computations must be carried out in a way that does not dominate the program execution time. Additionally, checksums should be completely independent of the main computation. Ideally, the asymptotic complexity of the checksum computations can be made smaller than the complexity of the main computation. Prior to our work, there has been no effort to generalize ABFT despite its effectiveness at detecting errors in *compute* that no fault tolerance schemes other than duplication can reliably detect. As we will show, the task of constructing and executing ABFT checksums is aptly suited to polyhedral analyses and specifically the analysis as it pertains to simplification. This is because ABFT checksum can be viewed as instances of reductions that can be simplified. In doing so we carry out an extensive analysis of stencil programs and provide a detailed comparison of three different design choices.

The contributions of this work can be summarized as follows: We provide an open-source implementaiton of the face lattice construction for arbitrary convex polyhedra leveraging the integer set library,"isl," [62]. We provide the first fully automatic push-button simplification engine as a stand-alone tool. We extend the simplification transformation to support a strictly more general class of programs than previously supported. We provide automation techniques for carrying out ABFT analyses in a compiler for a class of programs called stencil computations. We provide an in depth analysis of the design decision tradeoffs between ABFT schemes in stencil computations. We motivate the deeper connection between ABFT and the simplification transformation in the effort to remove the burden of analysis from the human and leverage the power of polyhedral compilation techniques.

Chapter 3

Background

In this work, we focus largely on programs that can be expressed within the polyhedral model, which encompasses a wide range of programs from scientific computing. In this chapter, we review the background on the polyhedral model and the simplification transformation.

3.1 Polyhedral program representation

The polyhedral model [16, 20, 25, 29, 6, 11, 13, 30, 22, 15, 17, 18, 23, 24, 10] is a mathematical formalism for reasoning about a precisely defined class of computations. It provides the technology to map high-level descriptions of compute- and data-intensive programs to a range of highly parallel targets. Polyhedral "programs" are most cleanly viewed as *equations* defined over *polyhedral domains*, evaluating an *expression* at each point therein. It enables us to reason precisely at the program statement instance level. The task of extracting the polyhedral representation (i.e., the set of affine recurrence equations) from a series of nested loops with affine control structure has been well studied [20].

The Alpha language [70, 14, 21, 33] is a high-level equational language that separates the specification of the program from its execution plan. The semantics of an Alpha program closely follows the program's equivalent equational representation. Given a set of affine recurrence equations, one can subsequently write the equivalent Alpha program in a straightforward manner. We do not review how to do this here and in this work, we assume that the input programs under study admit an equational representation. The Alpha grammar formally defines and supports many special types of nodes and expressions that may appear inthe abstract syntax tree (AST). We only review the key aspects below needed to make our discussion self-contained and provide a concrete example in Section 3.1.8. We use this representation heavily in subsequent chapters.

3.1.1 Polyhedra

Formally, a parametric integer polyhedron in $\mathbb{Z}^k$ for some $k \geq 0$ is defined as the intersection of a finite number of half-spaces and it may have one or more designated indices called size parameters. The polyhedron defined with m inequality constraints, n equality constraints, and q size parameters is written as,

$$\mathcal{D} = \{\ z \in \mathbb{Z}^k \ \mid\ Az + Bp \geq c;\ \ Cz + Dp = d;\ \} \tag{3.1}$$

where A is an $m \times k$ matrix, B is an $m \times q$ matrix, z is the k-vector $[i_1, i_2, ..., i_k]^T$, p is the q-vector $[p_1, p_2, ..., p_q]^T$, and c is the m-vector $[c_1, c_2, ..., c_m]^T$. Similarly, C is an $n \times k$ matrix, D is an $n \times q$ matrix, and d is the n-vector $[i_1, i_2, ..., i_n]^T$. The rows of A and B characterize the coefficients of inequalities for the indices (in z) and parameters (in p) and the rows of C and D characterize the equalities. Typically there is no upper bound on parameters, and a parametric polyhedron often defines an unbounded family of polytopes, one for each value of the parameters. The *volume*, cardinality, or the number of integer points, in such a parametric polyhedron is known to be a polynomial function of the parameters [37] and this polynomial represents the asymptotic complexity of a program that performs a constant time operation at every point in the polyhedron. The integer set library, `isl` [62], provides facilities for manipulating domains and can automatically deduce such polynomials from polyhedral loop programs. The degree of this polynomial, also called the number of *free dimensions*, or *dimensionality* of the polyhedron, is the number of indices in $\mathcal{D}$, less the number of its linearly independent equalities (i.e., $k - n$). It is the rank of the smallest linear subspace that spans the polyhedron. Note that the dimensionality of a particular domain becomes fuzzy when there are multiple size parameters. For example, when there are two size parameters, say M and N, the relative values of M and N affect whether or not M^2N is larger or smaller than MN^2. This is not a concern in our analysis, but something to keep in mind. All of the program variable domains in Alpha are represented as concrete `isl` objects.

3.1.2 Systems of Equations

Alpha [14, 21, 33] is an equational language and as such, at the top of the AST are ***system*** nodes. Intuitively, an Alpha system is defined as a "system of recurrence equations" over one or more variables scoped as inputs, outputs, and locals. Each variable v is defined over some domain, $\mathcal{D}_v$. For each output and local variable, there is exactly one equation that defines how each point in its domain, $\mathcal{D}_v$, is computed.

3.1.3 Equations and Expressions

Alpha equations are of the form below where Y is a program variable, E is an expression defined over the domain $\mathcal{D}_E$ and f is an affine function,

$$Y[f(z)] = E[z] \qquad \forall z \in \mathcal{D}_E$$

The expression E is evaluated at each point z in $\mathcal{D}_E$ and the answer defines the location in the variable Y specified by $f(z)$. Each Alpha expression is associated with two polyhedral domains. First, the ***expression domain*** of E is the domain over which E is well defined and is computed *bottom-up*. Next, the ***context domain*** of E is the domain over which it needs to be evaluated (i.e., where each value that contributes to an answer in the output needs to be written) and is computed *top-down*. Note that output and local variables may appear on either the left-hand-side or the right-hand-side of an equation. Input variables, however, may only appear on the right-hand-side (i.e., only read operations may be performed on input variables).

3.1.4 Dependence expressions

A ***dependence expression*** is an expression of the following form, where f is an affine function and z is a point in the expression domain of the expression E,

$$E[f(z)]$$

and should be understood as "the expression E evaluated at the point mapped from z by f" or equivalently as, "read the expression E at the point $f(z)$".

3.1.5 Restrict expressions

A ***restrict expression*** is an expression of the following form, where $\mathcal{D}$ is some domain,

$$\mathcal{D} : E$$

which should be read as "the expression E restricted to the subdomain $\mathcal{D}$". The expression domain of this restrict expression is the intersection of $\mathcal{D}$ with the expression domain of E.

3.1.6 Case expressions

Expressions can also be piecewise expressions defined over multiple semicolon-delimited disjoint pieces using ***case expressions***, which have the following form,

$$case\ \{\mathcal{D}_0 : E_0;\ \ \mathcal{D}_1 : E_1;\ \ \mathcal{D}_2 : E_2;\ \ ...\}$$

for any number of disjoint domains $\mathcal{D}_i$. Here, each of the pieces is itself a restrict expression.

3.1.7 Reduce expressions

Finally, the Alpha language also supports reduction expressions as first-class objects. Alpha ***reduce expressions*** generally have the following form, where Y is a program variable, E is the expression of the reduction body with the domain $\mathcal{D}_E$, and f_p is an affine function,

$$Y[f_p(z)] = \bigoplus E[z] \quad \forall z \in \mathcal{D}_E$$

where the expression E, evaluated at each point z in $\mathcal{D}_E$, is accumulated into the element of the variable Y at the location $f_p(z)$ by the $\oplus$ operator. The syntax for the reduce expression

that represents this is,

$$Y[f_p(z)] = \text{reduce}(\oplus, f_p, \mathcal{D}_E : E[z])$$

3.1.8 A concrete example

As an example, consider the triplely nested loops implementing the general matrix multiplication (GEMM) computation and its corresponding Alpha program representation shown in Figures 3.1 and 3.2 respectively.

```
// GEMM C-style loops

for (i=0; i<N; i++) {
    for (j=0; j<N; j++) {
        tmp = 0;
        for (k=0; k<N; k++) {
            tmp += A[i,k] * B[k,j];
        }
        D[i,j] = (alpha * tmp) + (beta * C[i,j]);
    }
}
```

Figure 3.1: C-style loops with static affine control

Note the layout, with the following main sections of code: inputs, outputs, locals, and equations. An Alpha program only specifies *what* is being computed and does not encode any explicit information about *when* particular values are computed nor *where* data is stored in memory. Explicit loops, as shown in Figure 3.1 for example, encode a particular scheduling of points in the computation domain. In many polyhedral analyses, the scheduling task is

```
affine gemm [N]->{:1<N}
    inputs
        alpha, beta : {}
        A, B, C : {[i,j] : 0<=i,j<N}
    outputs
        D : {[i,j] : 0<=i,j<N}
    let
        D[i,j] = alpha[]*reduce(+, [j,k], A[i,k]*B[k,j]) + beta[]*C[i,j];
.
```

Figure 3.2: Equivalent Alpha program representation

carried out separately as we discuss in the following section.

3.2 Scheduling

Given a set of program dependencies, typically in the form of a program dependence graph, a legal schedule is one that specifies an ordering among statements in the program such that all depednencies are satisfied. Scheduling is the process of associating each statement instance in the program with a timestamp such that all dependence relations are satisfied. We illustrate the idea by way of example. Consider the code snippet in Figure 3.3a that implements two matrix products, $C = AB$, and $F = DE$, followed by their point-wise sum, $G = C + F$ involving three loop nests and three statements denoted by the labels P, Q, and R. Most of our discussions in this dissertation involve an equational representation of programs, however we use concrete loops in the example here because it is easier to illustrate.

```
 0:      for (i=0; i<N; i++)
 1:        for (j=0; j<N; j++)
 2:          for (k=0; k<N; k++)
 3: P:         C[i][j] += A[i][k]*B[k][j];
 4:      for (i=0; i<N; i++)
 5:        for (j=0; j<N; j++)
 6:          for (k=0; k<N; k++)
 7: Q:         F[i][j] += D[i][k]*E[k][j];
 8:      for (i=0; i<N; i++)
 9:        for (j=0; j<N; j++)
10: R:       G[i][j] = C[i][j]+F[i][j];
```

(a) Sum of two matrix products with three separate loop nests, $G \leftarrow AB + DE$.

```
0:      for (i=0; i<N; i++)
1:        for (j=0; j<N; j++) {
2:          for (k=0; k<N; k++) {
3: P:         C[i][j] += A[i][k]*B[k][j];
4: Q:         F[i][j] += D[i][k]*E[k][j];
5:          }
6: R:       G[i][j] = C[i][j]+F[i][j];
7:        }
```

(b) One particular choice of loop fusion minimizing the read after write (RaW) distance, with a single loop nest.

Figure 3.3: Semantics preserving scheduling transformation

The core idea behind scheduling is that program statements are associated with domains characteristic of the sets of values the enclosing loop iterators can take, typically as sets of integer points. Program dependencies are represented as relations between these domains. Two statements are said to be in a dependence relation if they both access the same memory location and at least one of the accesses involves a write. A global program dependence

graph subsequently holds all pair-wise dependence relations between statement instances. Each statement is then associated with a scheduling function, typically denoted by Θ in the literature, that assigns each point in the statement's domain to a multidimensional timestamp such that all dependence relations in the global dependence graph are satisfied. For example, in Figure 3.3a the statements P and Q are defined over the 3-dimensional domains $\mathcal{D}_P = \mathcal{D}_Q = \{[i, j, k] : 0 \leq i, j, k < N\}$ and R over the 2-dimensional domain $\mathcal{D}_R = \{[i, j] : 0 \leq i, j < N\}$. Statements P and R are in a dependence relation since they each involve reads and writes to $C[i][j]$. Therefore, in any legal schedule, the timestamp of *each instance* of statement R must be strictly greater than the timestamp of the corresponding dependent *instance* of statement P. In other words, the condition $\Theta(z) > \Theta(z')$ must hold for all pairs of $z \in \mathcal{D}_R$ and $z' \in \mathcal{D}_P$ in the dependence graph between P and R. So the question becomes, given the space of all legal schedules encoded by the set of dependence relations in the global dependence graph, how is a particular *good* schedule chosen? This question is difficult because the space of all legal schedules can become intractably large.

Early works on multi-dimensional scheduling [25, 26, 9, 15] employed a greedy selection algorithm which was later formalized by Pouchet et al. [64] as a single convex integer linear programming (ILP) problem. Even though solving the ILP problem for arbitrary global dependence graphs is NP-complete, Pouchet et al. propose methods to prune the search space. From this, it is possible to employ optimizations such as minimizing read-after-write (RaW) dependence distances. Applying this optimization to the statements in Figure 3.3a for example, leads to the equivalent set of loops shown in Figure 3.3b. We can see that the distance (i.e., number of iterations) between the production of the value written into $C[i][j]$ on line 3 and when it is subsequently read on line 6 is at most N. Compare this to the distance between lines 3 and 10 in Figure 3.3a, which is on the order of N^3. Minimizing RaW distances is useful from a locality perspective and leads to good cache behavior because is it more likely for the value to be evicted from the cache over larger distances. N is still too large a distance and likely won't be optimal of course, but additional optimizations related

to locality, like loop tiling, are based on this same idea. This is just one particular example, there are many such objectives.

3.3 The Simplification Transformation

In this section, we review the simplification transformation, originally proposed by Gautam and Rajopadhye [49], which obtains algorithmic improvements in the input program's time complexity when particular conditions exist. In this work, we extend and improve this program transformation and therefore, this background is critical in order to understand our contributions. Generally, simplification operates on computations of the following form:

$$Y_{f_p(z)} \quad = \quad \bigoplus_{z \in \mathcal{D}} E_{f_d(z)} \tag{3.2}$$

where Y is an output variable, E is an arbitrary expression, and f_p and f_d are affine functions. We use the terminology of Gautam and Rajopadhye [49], summarized below:

- *Polyhedron*: A set of integer points defined by a finite list of inequality and equality constraints.
- *Reduction body* ($\mathcal{D}$): A d-dimensional polyhedron representing the values of the program variable indices involved in the reduction's accumulation.
- *Facet (or face)*: A k-dimensional face of the reduction body described uniquely by a subset of its inequality constraints treated as equalities.
- *Face lattice*: The hierarchical arrangement of faces of the reduction body.
- *Write function* (f_p): A rank-deficient affine map from $\mathbb{Z}^d \rightarrow \mathbb{Z}^{d-a}$ defining to which element of the output each point in the reduction body accumulates.[1]
- *Accumulation space* ($\mathcal{A}$): The a-dimensional space characterized by the null space of the write function.

[1] Rank-deficiency implies that $a > 0$.

- *Read function* (f_d): A (potentially rank-deficient) affine map from $\mathbb{Z}^d \to \mathbb{Z}^{d-r}$ characterizing from which element of the input each point in the reduction body reads.

- *Reuse space* ($\mathcal{R}$): The r-dimensional space characterized by the null space of the read function.

- *Reuse vector* ($\vec{\rho}$): Any vector in the reuse space.

Such programs consist of arbitrarily nested loops with affine control and a single statement in the body, plus initialization statements as appropriate. Each statement accumulates values into some element of an output array using the reduction operator $\oplus$. The right-hand sides of the statements are arbitrary expressions evaluated in constant time, reading other array variables via affine access functions. Our reduction operators are associative and commutative, so the execution order of the loops is irrelevant. Every instance of the loop body (for every legal value of the surrounding indices) can be treated as a new dummy variable. Simplification is possible when the reuse space is non-empty (i.e., the read function, f_d, is rank-deficient).

3.3.1 Face Lattice

The face lattice [37] is an important data structure for simplification. The face lattice of a polyhedron $\mathcal{D}$ is a graph whose nodes are the *facets* of $\mathcal{D}$. Each face in the lattice is the intersection of $\mathcal{D}$ with one or more *equalities* of the form $\alpha z + \gamma = 0$ for $z \in \mathcal{D}$ obtained by *saturating* one or more of the inequality constraints in $\mathcal{D}$. We refer to each k-dimensional face as a *(k)-face*.

More than one constraint may be saturated to yield recursively, facets of facets, or *faces*. In the lattice, faces are arranged level by level, and each face saturates exactly one constraint in addition to those saturated by its immediate ancestors.

Thick and Extended Faces

Gautam and Rajopadhye [49] introduce the notion of a thick face and what they call an *effectively saturated constraint*. An effectively saturated constraint is either a single equality constraint or a pair of parallel inequality constraints separated by a constant, which should be viewed as a single "thick equality" constraint.

For example, consider the following domains:

$$\mathcal{D}_1 = \{[i,j] \mid (0 \leq i < N) \wedge (0 \leq j < N)\} \tag{3.3}$$

$$\mathcal{D}_2 = \{[i,j] \mid (0 \leq i < 10) \wedge (0 \leq j < N)\} \tag{3.4}$$

The cardinalities of $\mathcal{D}_1$ and $\mathcal{D}_2$ are polynomials of the size parameter N. There are N^2 points in $\mathcal{D}_1$ and there are $10N$ points in $\mathcal{D}_2$. We say that $\mathcal{D}_1$ is 2-dimensional because the cardinality of $\mathcal{D}_1$ is quadratic in N. Similarly, we view $\mathcal{D}_2$ as 1-dimensional because its cardinality is linear in N, and we say that the consraint $i = 0$ is effectively saturated. The dimensionality of an arbitrary polyhedron $\mathcal{D}$ is said to be its number of indices less its number of effectively saturated constraints.

An Example

The face lattices for $\mathcal{D}_1$ and $\mathcal{D}_2$, from Equations 3.4 and 3.3, are shown on the left and right respectively of Figure 3.4. The four constraints of $\mathcal{D}_1$, $0 \leq i$, $0 \leq j$, $i < N$, and $j < N$, are denoted as 0, 1, 2, and 3. Its four edges (or 1-faces) are described by saturating each constraint. For example, the node "{0}" denotes saturing the 0'th constraint, $0 \leq i$, and represents the edge at $i = 0$. Similarly, the node "{0,1}" represents saturating both constraints, $0 \leq i$ and $0 \leq j$, and therefore represents the vertex at $[0, 0]$.

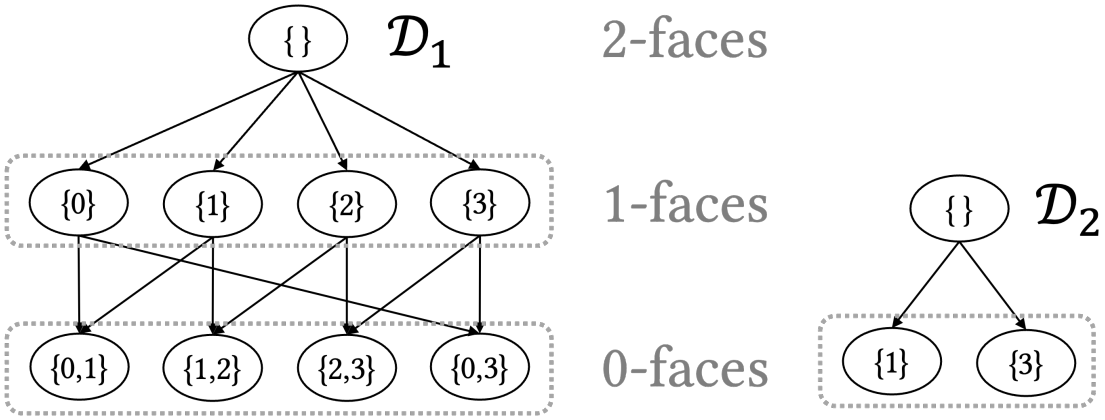


Figure 3.4: Face lattice of $\mathcal{D}_1$ in Equation 3.3, the square with four edges (1-faces) and four vertices (0-faces) is shown on the left and $\mathcal{D}_2$ in Equation 3.4, the thick line segment with two vertices (0-faces) on the right.

3.3.2 Single-Step Simplification

Consider the following example which computes an N-element array Y:

$$Y_i = \sum_{j=0}^{i} X_{i-j} \tag{3.5}$$

The equation has a reduction with the addition operator. The reduction body is defined over the domain $\mathcal{D} = \{[i, j] \mid 0 \leq j \leq i < N\}$, the red triangle in Fig. 3.5.

The i'th result, Y_i, is the accumulation of the values of the reduction body at points in the i'th diagonal of the triangle. The complexity of the equation is the number of integer points in $\mathcal{D}$, namely $O(N^2)$. The reuse space is the null space of the indexing expression appearing inside the reduction body, which is $f_d : \{[i, j] \to [i - j]\}$ in this example. Simplification is possible because the body has redundancy along the vector $\rho = [1, 1]$ (green arrow). Any two points $[i, j], [i', j'] \in \mathcal{D}$ separated by a scalar multiple of ρ read the same value of X because $X_{i-j} = X_{i'-j'}$. In other words, the body expression evaluates to the same value at all points along $[1, 1]$. Simplification exploits this reuse to read and compare only the $O(N)$ distinct values. A geometric explanation of simplification is:

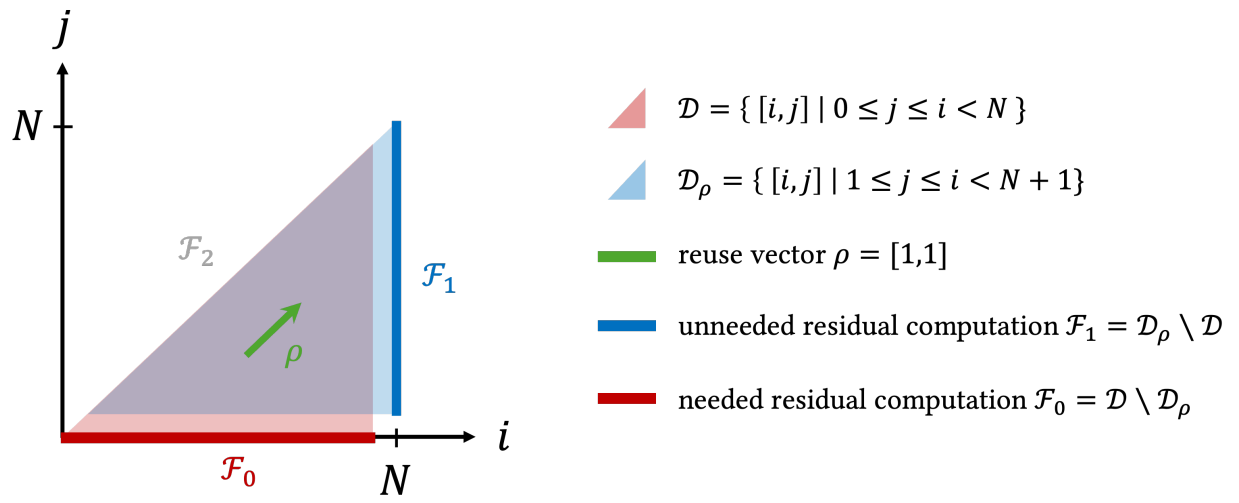


Figure 3.5: Simplification of a quadratic equation (computation defined over a triangle) to a linear complexity (the residual computation is defined only over the points in the bottom row).

- Translate $\mathcal{D}$ (shift the red triangle) by ρ (green arrow) yielding $\mathcal{D}_\rho$ (the blue triangle).
- Ignore all computations in the intersection of the two, $\mathcal{D} \cap \mathcal{D}_\rho$.
- Evaluate the residual computation on only the facets (here, edges $\mathcal{F}_i$) of $\mathcal{D}$ and/or $\mathcal{D}_\rho$.

Additionally, some of these facets can be ignored: the diagonal one $\mathcal{F}_2$, being parallel to ρ, was already included in the intersection, and the vertical (blue) one $\mathcal{F}_1$ is *external* since it does not contribute to any answer. This leaves a residual computation on only the bottom facet $\mathcal{F}_0$. Thus, we can replace the $O(N^2)$ equation by Equation 3.6, which has only $O(N)$ complexity.

$$Y_i = \begin{cases} X_i & \text{if } i = 0 \\ Y_{i-1} + X_i & \text{if } i > 0 \end{cases} \tag{3.6}$$

Indeed, Equation 3.5 specifies a simple scan (prefix-sum) of the input, albeit counter-intuitively.

3.3.3 Equivalence Partitioning of Infinite Choices

Notice that Equation 3.6 expresses Y_i in terms of Y_{i-1}. It is equally possible to simplify Equation 3.5 by expressing Y_i in terms of Y_{i+1} instead. There are two choices because the reuse space in this example is 1-dimensional (with two directions), and we could choose to exploit reuse along the vector $\rho_1 = [1, 1]$ or its negation, $\rho_2 = [-1, -1]$. In the general case, when the reuse space is 2-dimensional or higher, the number of candidate choices is infinite.

However, not all choices of reuse need to be explored. We illustrate this concretely in Section 4.3.1, but the intuition is that any two reuse vectors resulting in the same combination of residual computation can be viewed as the same candidate choice. Gautam and Rajopadhye [49] refers to the subset of ρ_i that results in the same combination of residual computation as an *equivalance class*. Then, the set of all equivalence classes can be explored with dynamic programming. As long as a single reuse vector from each equivalence class is considered, this is sufficient to guarantee the final simplified program's optimality.

3.3.4 Recursive Simplification

In the general case, reductions have a d-dimensional reduction body, an a-dimensional accumulation, and an r-dimensional reuse space. The process of Section 3.3.2 is applied recursively on the face lattice, starting with $\mathcal{D}$. At each step, we simplify the facets of the current face $\mathcal{F}$. The key idea is that exploiting reuse along ρ avoids evaluating the reduction expression at most points in $\mathcal{F}$. Specifically, let $\mathcal{F}'$ be the translation of $\mathcal{F}$ along ρ. Then all the computation in $\mathcal{F} \cap \mathcal{F}'$ is avoided, and we only need to consider the two differences $\mathcal{F}' \backslash \mathcal{F}$ and $\mathcal{F} \backslash \mathcal{F}'$, i.e., the union of some of the facets of $\mathcal{F}$. We are left with *residual computations* defined only on (a subset of) the (thick/extended) facets of $\mathcal{F}$.

At each step of the recursion, the asymptotic complexity is reduced by exactly one polynomial degree, as facets of $\mathcal{F}$ are strictly smaller dimensional subspaces. Furthermore, at each step, the newly chosen ρ is linearly independent of the previously chosen ones. Hence, the method is optimal—all available reuse is fully exploited. This holds regardless of the

choice of ρ at any level of the recursion, even though there may be infinitely many choices. We will walk through additional concrete examples as needed in subsequent chapters.

Although simplification is a powerful program transformation, it is incomplete in several ways. Gautam and Rajopadhye's 2006 result is primarily theoretical, and they omitted several crucial details required to implement a simplifier in practice. Specifically, they do not give methods for constructing the set of equivalence classes described in Section 3.3.3. Subsequently, the number of reuse vectors within a particular equivalence class is generally infinite, and we must still select one. We address both of these concerns next in Chapter 4. Furthermore, the simplification algorithm does not always succeed, particularly when the reduction operator does not admit an inverse (i.e., min or max). Some choices of reuse may require removing previously accumulated partial answers (e.g., subtraction if the reduction operator is addition). It may be the case that all possible choices of reuse require such operations, in which case simplification fails despite the existence of unecessary redundancy in the computation. We discuss this aspect in more detail and propose solutions in Chapter 5, showing how to guarantee that simplification always succeeds.

Chapter 4

Simplification of Polyhedral Reductions in Practice

4.1 Introduction

In this chapter, we study the optimization of programs that can be specified by reductions within the polyhedral model. Reductions are ubiquitous in computing and typically involve applying an associative, often commutative, operator to collections of inputs to produce collections of results. Such operations are interesting because they often require special handling to obtain good performance. The OpenMP C/C++ multithreading API [88] even has directives tailored specifically for parallelizing reductions. Typical compiler optimizations yield, at best, a constant fold speedup. However, in some cases, the input program specification may involve *reuse*—the same value contributing to multiple results. When properly exploited, it is possible to improve the asymptotic complexity of such programs. A complexity improvement from, say, $O(N^3)$ to $O(N^2)$ yields unbounded speedup since, asymptotically, N can be arbitrarily large. Optimizations of this type have traditionally relied on very clever and manual human analysis and engineering effort.

Gautam and Rajopadhye [49] previously showed how to reduce, by such polynomial degrees, the asymptotic complexity of polyhedral reductions. They developed a program transformation called *simplification* and outlined, without providing all necessary details, a recursive algorithm to automatically simplify polyhedral reductions. Their work is purely theoretical and omits details needed to implement a simplifier. In this chapter, we address these issues, and in doing so, make the following contributions:

- Our implementation includes a heuristic algorithm for choosing the specific reuse to exploit at each step of the recursion, thereby realizing a complete push-button simplification engine.

- Using the integer set library (`isl`) [62], we provide an open source implementation to construct the *face lattice* of arbitrary parameterized convex polytopes. This is a critical data structure for simplification but has other uses beyond that, notably in program verification [91, 79].

We evaluate the effectiveness of our approach on a range of programs involving reductions and provide an accompanying software artifact as a proof of concept. Interestingly, our simplification of an $O(N^4)$ RNA secondary structure prediction algorithm produces *four* distinct versions, all with cubic complexity. One of these rediscovers the 1999 result of Lyngsø et al. (called the "fast-i-loops" algorithm/technique), while the other three were previously unknown.

The remainder of the chapter is organized as follows. In Section 4.2, we provide a concrete example that motivates our work. In Section 4.3, we review how we address the practical issues left open by Gautam and Rajopadhye [49], as discussed previously in Section 3.3, and explain the details of the implementation of our compiler. We provide an analytical and quantitative evaluation in Sections 4.4 and 4.5. Finally, we discuss open problems and conclude in Sections 4.6 and 4.7.

4.2 Motivating Example

This section summarizes the fast-i-loops algorithm, a real-world non-trivial example of a reduction which has previously only ever been optimized manually. Ribonucleic acid (RNA) forms one of the building blocks of life. It is described as a sequence of bases drawn from a 4-letter alphabet. An important determiner of the molecule's function is its *secondary structure*, which indicates the bases that are bonded together. One method for secondary structure prediction uses thermodynamic equations to determine the minimum free energy configuration (i.e., the most likely to occur in nature). These equations are defined as a system of dynamic programming recurrence equations. The original definition [4] populates these tables with a time complexity of $O(N^3)$, with one notable exception that calculates

the energy of an *internal loop* structure in $O(N^4)$ time. Equation 4.1 is a representative example of the same form as this equation. It computes a 2-dimensional result, $Y_{i,j}$, as the minimum of all pairs of points between i and j.

$$Y_{i,j} = \min_{i<p<q<j} \left(A_{p,q} + B_{p-i+j-q} + C_{|p-i-j+q|} \right) \tag{4.1}$$

In 1999, Lyngsø et al. [38] exploited an invariant property in Equation 4.1 to rewrite it essentially as Equations 4.2 and 4.3, which have $O(N^3)$ complexity. A derivation of how our simplifier produces this *fast-i-loops* algorithm is in Section 4.5.

$$Y_{i,j} = \min_{1<k<j-i} \left(B_k + Z_{i,j,k} \right) \tag{4.2}$$

$$Z_{i,j,k} = \min \begin{pmatrix} A_{i+1,j-k+1} + C_{|-k+2|} \\ A_{i+k-1,j-1} + C_{|k-2|} \\ Z_{i+1,j-1,k-2} \end{pmatrix} \tag{4.3}$$

Since its discovery, there have been several implementations of the fast-i-loops algorithm [44, 83, 61]. Each implementation required painstaking effort by the authors to first transform the specification into a more usable format and then write the code that implements the algorithm. Our compiler is able to take the original $O(N^4)$ specification of Equation 4.1 and automatically discover four $O(N^3)$ improved algorithms.

4.3 Simplification in Practice

In this section, we discuss methods for constructing the set of equivalence classes described in Section 3.3.3 and selecting a candidate reuse vector from each.

4.3.1 Constructing Equivalence Classes

Simplification proceeds recursively down the face lattice, simplifying one facet at a time. Let $\mathcal{F}$ be an arbitrary facet of the reduction body. Let $\mathcal{F}_i$ denote the i'th facet of $\mathcal{F}$ (i.e., its i'th child), and let ν_i be the linear part of the normal vector of $\mathcal{F}_i$. Let the symbol $\oplus$ be the reduction operator, and $\ominus$ be its inverse if $\oplus$ is invertible. The single-step simplification of $\mathcal{F}$, from Section 3.3.2, with the reuse vector ρ, results in a residual computation on some of its facets. The orientation of ρ relative to each facet $\mathcal{F}_i$ dictates the type of residual computation that occurs on $\mathcal{F}_i$. There are three possibilities, depending on the sign of the dot product between ρ and ν_i:

1. If $\rho \cdot \nu_i > 0$ then $\mathcal{F}_i$ contributes with the $\oplus$ operator.

2. If $\rho \cdot \nu_i < 0$ then $\mathcal{F}_i$ contributes with the $\ominus$ operator.

3. If $\rho \cdot \nu_i = 0$ then $\mathcal{F}_i$ does not contribute at all.

Each facet $\mathcal{F}_i$ may be labeled as either an $\oplus$-face, $\ominus$-face, or $\oslash$-face respectively. We say that ρ *induces a particular labeling*, $\mathcal{L}$, on the facets of $\mathcal{F}$. Each labeling corresponds to one of the equivalence classes. Given a particular labeling $\mathcal{L}$, we can construct the set of ρ that induce $\mathcal{L}$ as the following:

$$\mathcal{R}_{\mathcal{L}} = \mathcal{R} \cap \{\rho \mid (\rho \cdot \nu_i > 0) \ \wedge \ (\rho \cdot \nu_j < 0) \ \wedge \ (\rho \cdot \nu_k = 0)\} \tag{4.4}$$

where $\mathcal{R}$ is the overall reuse space and ν_i denotes the normal vectors of facets $\mathcal{F}_i$ labeled as $\oplus$-faces, ν_j as $\ominus$-faces, and ν_k as $\oslash$-faces. In other words, we further constrain $\mathcal{R}$ to isolate the set of ρ that induces the labeling $\mathcal{L}$.

Since each facet $\mathcal{F}_i$ of a face $\mathcal{F}$ can be given one of three labels, there are a total of 3^m labelings. Our implementation currently enumerates all them, but there is a possibility for optimization (see Figure 4.1 which has only 12 possibilities rather than 27). Some are impossible, e.g., there is no way to mark all facets with the same label.

An Example

Consider the reduction in the following equation:

$$Y_i = \sum_{(j,k)\geq 0}^{j\leq i \,\wedge\, k\leq i-j} X_k \tag{4.5}$$

for $0 \leq i \leq N$. The domain of the reduction body, $\mathcal{D}$, is the tetrahedron:

$$\{[i,j,k] \mid (0 \leq i \leq N) \wedge (0 \leq j) \wedge (k \leq i-j) \wedge (0 \leq k)\} \tag{4.6}$$

The dependence function, f_d, in this example is the mapping:

$$\{[i,j,k] \rightarrow [k]\} \tag{4.7}$$

and thus the reuse space, $\mathcal{R}$, is the ij-plane:

$$\{[i,j,k] \mid k = 0\} \tag{4.8}$$

The tetrahedron $\mathcal{D}$ has four 2-faces. The 2-face in the ij-plane (characterized by $k = 0$) will always be labeled as a $\oslash$-face regardless of the choice of reuse, as any vector in the ij-plane is orthogonal to the normal vector of this face. There are a total of 12 ways to label the remaining three 2-faces, illustrated in Figure 4.1.

4.3.2 Selecting Candidate Reuse Vectors

Given a particular labeling, $\mathcal{L}$, the set of reuse vectors that induce the labeling, $\mathcal{R}_\mathcal{L}$, per Equation 4.4 is either empty or unbounded. From the perspective of a single-step simplification, there is no reason to prefer one $\rho \in \mathcal{R}_\mathcal{L}$ over another. Any ρ results in a decrease in asymptotic complexity by one polynomial degree, which follows from the optimality claim of prior work on simplification. Therefore, we employ a simple heuristic that selects the

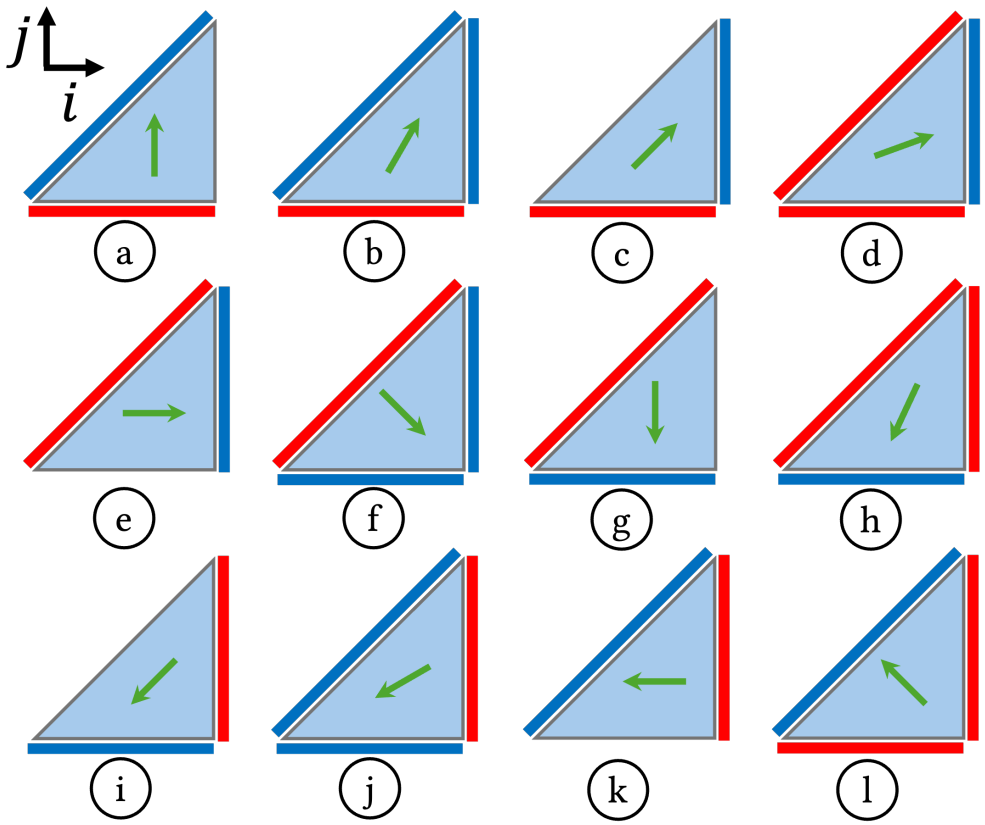


Figure 4.1: Equivalence classes explored for the single-step simplification of Equation 4.5, with $\oplus$-faces highlighted in red, $\ominus$-faces in blue, and $\oslash$-faces not highlighted.

smallest integer point closest to the origin. Whether or not a ρ can be chosen to guarantee that the constant factor is minimized remains an open problem.

An Example

The subsets of the reuse space that induce the 12 labelings illustrated in Figure 4.1 are given below, per the formulation in Equation 4.4:

$$\mathcal{R}_a = \{[i,j,k] \mid (k=0) \wedge (-i=0) \wedge (j>0) \wedge (i-j<0)\}$$

$$\mathcal{R}_b = \{[i,j,k] \mid (k=0) \wedge (-i<0) \wedge (j>0) \wedge (i-j<0)\}$$

$$\mathcal{R}_c = \{[i,j,k] \mid (k=0) \wedge (-i<0) \wedge (j>0) \wedge (i-j=0)\}$$

$$\mathcal{R}_d = \{[i,j,k] \mid (k=0) \wedge (-i<0) \wedge (j>0) \wedge (i-j>0)\}$$

$$\mathcal{R}_e = \{[i,j,k] \mid (k=0) \wedge (-i<0) \wedge (j=0) \wedge (i-j>0)\}$$

$$\mathcal{R}_f = \{[i,j,k] \mid (k=0) \wedge (-i<0) \wedge (j<0) \wedge (i-j>0)\}$$

$$\mathcal{R}_g = \{[i,j,k] \mid (k=0) \wedge (-i=0) \wedge (j<0) \wedge (i-j>0)\}$$

$$\mathcal{R}_h = \{[i,j,k] \mid (k=0) \wedge (-i>0) \wedge (j<0) \wedge (i-j>0)\}$$

$$\mathcal{R}_i = \{[i,j,k] \mid (k=0) \wedge (-i>0) \wedge (j<0) \wedge (i-j=0)\}$$

$$\mathcal{R}_j = \{[i,j,k] \mid (k=0) \wedge (-i>0) \wedge (j<0) \wedge (i-j<0)\}$$

$$\mathcal{R}_k = \{[i,j,k] \mid (k=0) \wedge (-i>0) \wedge (j=0) \wedge (i-j<0)\}$$

$$\mathcal{R}_l = \{[i,j,k] \mid (k=0) \wedge (-i>0) \wedge (j>0) \wedge (i-j<0)\}$$

The second piece, "$(-i...)$", in each of these comes the linear part of the constraint $N-i \geq 0$ from Equation 4.6. Similarly, the "$(i-j...)$" pieces should really be understood as $i-j-k...$ from Equation 4.6, but we drop k to save space since we also have the constraint $k=0$. The constraints are colored to correspond with the edge colorings in Figure 4.1. Our heuristic selection criteria will choose $\rho_a = [0,1,0]$ from the first one, $\rho_b = [1,2,0]$ from the second one, etc., and $\rho_l = [-1,1,0]$ for the last since these are the smallest integer points closest to the origin.

4.3.3 Implementation Details

We implemented the simplification algorithm in the Alpha language [14, 21] and the AlphaZ system [70], which supports reduction operations as first-class objects. This provides concrete representations of the reduction body, projection function, dependence function, and face lattice described in Section 3.3 using `isl` (the Integer Set Library [62]) objects. The semantics of an Alpha program closely follows the program's equivalent equational represen-

tation. The workflow of our tool-chain is illustrated in Figure 4.2. Given an input equational program specification, our compiler runs the 2006 simplification algorithm and produces a series of transformed programs, one for each simplification possible. We then generate C code for each transformed program to be compiled and run.

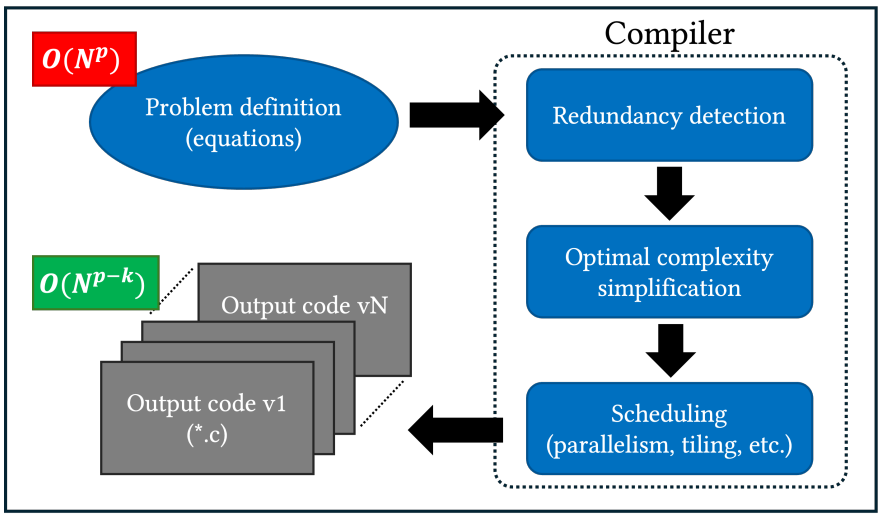


Figure 4.2: Alpha to C compilation pipeline.

Consistent with Gautam and Rajopadhye [49], we assume that the input program admits an equational representation (i.e., can be specified in the Alpha language) and that the program involves reduction operations. An imperative polyhedral loop program can be easily brought into such a form (e.g., see the pioneering work of Feautrier [20, 31, 42] on how this could be done). We also assume that programs involve a single size parameter.

4.4 Case Studies

In this section, we evaluate the efficacy of our implementation based on the design choices described in Section 4.3. In the following sections, we introduce several examples and confirm that our implementation can find the corresponding simplifications. In all examples discussed below, we confirmed that the generated code produces the correct answers and exhibits the expected reduced asymptotic complexity.

Table 4.1 shows the initial runtime complexity and the expected complexity after sim-

plification for each tested program. We have created a specification for each example and confirmed that our compiler produces the expected simplifications. We provide the original and simplified specifications with their corresponding generated C code in an accompanying artifact.

Table 4.1: Program simplification starting and expected asymptotic runtime complexities

Program	Starting Complexity	Final Complexity
Recursive Simplification	$O(N^3)$	$O(N)$
Reduction Decomposition	$O(N^3)$	$O(N^2)$
Distributivity	$O(N^3)$	$O(N^2)$

4.4.1 Recursive Simplification

Programs with multiple dimensions of reuse require recursive simplification (from Section 3.3.4) to produce an optimal specification. To test our implementation of this, consider the example from Section 4.3.1, given again in Equation 4.9.

$$\begin{gathered} Y_i = \sum_{\{j,k\} \in \mathcal{D}} A_k \\ \mathcal{D} = \{[j,k] \mid (0 \leq j \leq i) \wedge (0 \leq k \leq i-j)\} \end{gathered} \tag{4.9}$$

A naive execution of this specification has a time complexity of $O(N^3)$. However, this equation specifies a prefix scan of a prefix scan. This can be performed in linear time by performing a prefix scan over our inputs, saving these results in a temporary variable, then performing a prefix scan over the temporary variable. This is demonstrated by Equations

4.10 and 4.11:

$$Y_i = \begin{cases} Z_0 & \text{if } i = 0 \\ Z_i + Y_{i-1} & \text{if } i > 0 \end{cases} \tag{4.10}$$

$$Z_i = \begin{cases} A_0 & \text{if } i = 0 \\ A_i + Z_{i-1} & \text{if } i > 0 \end{cases} \tag{4.11}$$

Each application of simplification is only able to lower the asymptotic time complexity by one degree. Thus, to get our time complexity down from $O(N^3)$ to $O(N)$, we will need to apply simplification at least twice via recursive simplification. Thus, our compiler being able to produce any correct linear-time programs verifies a correct implementation of this feature.

4.4.2 Reduction Decomposition

Recall from Section 3.3, the accumulation space of a reduction may be multidimensional. Gautam and Rajopadhye [49] proposed several simplification-enhancing transformations that can be used to expand the reuse space. Given two functions f''_p and f'_p such that $f_p = f''_p \circ f'_p$, a reduction of the form in Equation 3.2 with multi-dimensional accumulation may be rewritten as the following two reductions,

$$Y_{f''_p(z)} = \bigoplus_{z \in \mathcal{D}} Z_{f'_p(z)} \tag{4.12}$$

$$Z_{f'_p(z)} = \bigoplus_{z \in \mathcal{D}} X_{f_r(z)} \tag{4.13}$$

with the introduction of a new variable Z to hold partial answers. We can think of this as decomposing a higher dimensional reduction into a lower dimensional *reduction of reductions*, which is legal because the order of accumulation does not matter.

Consider the example of Equation 4.14:

$$\begin{gathered} Y_i = \max_{\{j,k\} \in \mathcal{D}} A_{j,k} \\ \mathcal{D} = \{[j,k] \mid (i \leq j \leq 2i) \wedge (i \leq k \leq 3i - j)\} \end{gathered} \tag{4.14}$$

While there is reuse along the i dimension (since we're reading $A_{j,k}$), we are not able to perform simplification. All possible labelings of the face lattice involve at least one necessary $\ominus$-face, meaning the transformation requires the nonexistent inverse of the max operation.

However, we can perform reduction decomposition to expose reuse that can lead to simplification. This is done by performing a change of basis to introduce $m = j + k$ and replace all instances of j accordingly. We then rewrite this single 2-dimensional reduction into two 1-dimensional reductions such that the outer reduction is over m and the inner is over k. Performing this transformation, then isolating the inner reduction to its own variable, produces Equation 4.15:

$$Y_i = \max_{m=2i}^{3i} Z_{i,m} \qquad\qquad Z_{i,m} = \max_{k=i}^{m-i} A_{m-k,k} \tag{4.15}$$

The equation for Z can now be simplified where $Z_{i,m}$ is computed from $Z_{i+1,m}$, producing an $O(N^2)$ program per Equation 4.16:

$$Z_{i,m} = \max\big(Z_{i+1,m},\ A_{m-i,i},\ A_{i,m-i}\big) \tag{4.16}$$

This is the only such program, and our compiler is able to find it automatically. The program specification was tested and confirmed to compute the correct answers. Thus, we can confirm the correct implemenation of reduction decomposition to enable simplification.

4.4.3 Distributivity

Consider the case where a reduction's body contains a binary operation that distributes over the reduction operator (e.g., multiplication inside a summation). If one of the terms is

invariant at all points within the reduction body, it can be factored out. This in turn may expand the reuse space of the reduction, enabling simplification.

Equation 4.17 below, which performs a summation over the product of two terms, demonstrates such a situation:

$$\begin{gathered} Y_i = \sum_{\{j,k\} \in \mathcal{D}} (A_{i,j+k} \times B_{k,j}) \\ \mathcal{D} = \{[j,k] \mid (0 \leq j \leq i) \wedge (0 \leq k \leq i)\} \end{gathered} \tag{4.17}$$

When computing a particular Y_i, all points in the reduction where $j + k$ are the same will read the same value of A. With the summation as written, we cannot exploit this. However, we can apply the same reduction decomposition transformation as in Section 4.4.2 to help us expose the reuse. Performing the change of basis to introduce $m = j + k$ and replace j such that the outer sum is over m results in Equation 4.18:

$$Y_i = \sum_{m=0}^{2i} \left(\sum_{k=\max(0,m-i)}^{\min(i,m)} (A_{i,m} \times B_{m-k,k}) \right) \tag{4.18}$$

Now, notice that $A_{i,m}$ is invariant at all points of the inner summation over k. We can exploit the fact that multiplication distributes over addition to pull this term out, then isolate the inner reduction to its own variable, producing Equation 4.19:

$$Y_i = \sum_{m=0}^{2i} A_{i,m} \times Z_{i,m} \qquad\qquad Z_{i,m} = \sum_{k=\max(0,m-i)}^{\min(i,m)} B_{m-k,k} \tag{4.19}$$

Simplification can be applied to $Z_{i,m}$, allowing each value to be computed in constant time. There are two choices for how to reuse answers: either $Z_{i+1,m}$ or $Z_{i-1,m}$. Our compiler is able to produce both of these programs automatically, and both specifications were found to compute the correct answers. Thus, we can confirm that our compiler can correctly exploit the distributive property to enable simplification.

4.5 Evaluation

In this section, we will evaluate our compiler's ability to automatically reproduce the same *fast-i-loops* algorithm that Lyngsø et al. discovered. To do this, we will first show how the improved algorithm can be derived by hand. Then, we will briefly discuss our compiler's rediscovery of this algorithm, plus three alternative algorithms produced from the same specification. Finally, we will empirically confirm that the simplified programs produce the correct results and that they exhibit the expected asymptotic runtime characteristics.

4.5.1 Reproducing the Fast-i-Loops Algorithm

Recall the $O(N^4)$ Equation 4.1 for computing the energy of an interior loop structure (used in RNA secondary structure prediction) from Section 4.2, repeated here for convenience.

$$Y_{i,j} = \min_{i<p<q<j} \big(A_{p,q} + B_{p-i+j-q} + C_{|p-i-j+q|}\big) \tag{4.20}$$

First, we can decompose this reduction by introducing $k = p - i + j - q$ (the indexing expression for B), replacing p accordingly. We let the outer minimization be over k and the inner be over q.

$$Y_{i,j} = \min_{2\leq k<j-i} \left(\min_{j-k<q<j} \big(A_{i-j+k+q,q} + B_k + C_{|k-2j+2q|}\big) \right) \tag{4.21}$$

Notice that the B_k term is now invariant within the inner minimization and thus can be factored out. We can then isolate the inner minimization to its own variable.

$$Y_{i,j} = \min_{2\leq k<j-i} \big(B_k + Z_{i,j,k}\big) \tag{4.22}$$

$$Z_{i,j,k} = \min_{j-k<q<j} \big(A_{i-j+k+q,q} + C_{|k-2j+2q|}\big) \tag{4.23}$$

Finally, we can notice that $Z_{i+1,j-1,k-2}$ minimizes a subset of the terms that $Z_{i,j,k}$ does.

We can then apply simplification to rewrite $Z_{i,j,k}$ as the minimum of $Z_{i+1,j-1,k-2}$ and a constant number of additional points in the reduction body. This gives us our desired $O(N^3)$ algorithm, repeated below.

$$Z_{i,j,k} = \min \begin{pmatrix} A_{i+1,j-k+1} + C_{|-k+2|} \\ A_{i+k-1,j-1} + C_{|k-2|} \\ Z_{i+1,j-1,k-2} \end{pmatrix} \tag{4.24}$$

4.5.2 Newly Discovered Algorithms

We developed a specification for the full $O(N^4)$ fast-i-loops algorithm, matching the equation format and variable names presented by Jacob et al. [61]. When given to our compiler, it was able to automatically discover this same algorithm that Lyngsø et al. described, plus three alternative algorithms which were previously unknown. The specification and the code to derive all four simplified algorithms can be found in the accompanying artifact.

One of the newly discovered algorithms performs a different decomposition using the expression used to index C instead of B. That is, it introduced $l = p - i + q - j$. Since this value may be negative (as $q < j$), and since its absolute value is used to access C, different answers must be reused if l is positive or negative. Equations 4.25 and 4.26 below are representative of the algorithm produced by our compiler. We refer to this algorithm as the "New Algorithm".

$$Y_{i,j} = \min_{i-j+3 \le l \le j-i-3} \left(C_{|l|} + Z'_{i,j,l} \right) \tag{4.25}$$

$$Z'_{i,j,l} = \begin{cases} \min \begin{pmatrix} Z'_{i-1,j-1,l+2} \\ A_{i+l+1,j-1} + B_{l+2} \end{pmatrix} & \text{if } l \ge 0 \\ \min \begin{pmatrix} Z'_{i+1,j+1,l-2} \\ A_{i+1,j+l-1} + B_{-l+2} \end{pmatrix} & \text{if } l < 0 \end{cases} \tag{4.26}$$

The final two algorithms are a hybrid approach. In short, they split the minimization of

Table 4.2: Polynomials for the total number of loop iterations in each RNA minimum free energy algorithm.

Program	Loop Iterations
Original Algorithm	$\frac{1}{24}N^4 + \frac{1}{12}N^3 - \frac{1}{24}N^2 + \frac{35}{12}N - 1$
Lyngsø Algorithm	$N^3 - 5N^2 + 17N - 30$
New Algorithm	$\frac{4}{3}N^3 - 8N^2 + \frac{62}{3}N - 14$
Hybrid 1	$\frac{13}{12}N^3 - \frac{43}{8}N^2 + \frac{271}{24}N + \frac{1}{4}\lfloor\frac{N}{2}\rfloor - 3$
Hybrid 2	$\frac{5}{4}N^3 - \frac{61}{8}N^2 + \frac{211}{8}N - \frac{1}{4}\lfloor\frac{N}{2}\rfloor - 41$

$Y_{i,j}$ into two cases early on, based on the positive and negative values of $l = p - i + q - j$. One of the splits use the Lyngsø simplification described in Section 4.5.1, while the other split uses the alternative simplification of Equations 4.25 and 4.26. These two algorithms are referred to as "Hybrid 1" and "Hybrid 2".

Our compiler produces polynomials for the total number of loop iterations performed by each program as a metric for estimating the performance of each algorithm. These are listed in Table 4.2.

4.5.3 Empirical Verification of Expected Complexities

The original $O(N^4)$ algorithm and all four $O(N^3)$ algorithms were used to generate single threaded, naive, demand-driven C code [35]. All of the programs were compiled by GCC using the Makefile provided by the code generator. We tested the produced executables to empirically verify:

- The simplified versions produce the same results as the $O(N^4)$ original program.
- The asymptotic time complexity of each program is as expected.

Correctness of the results was determined by using the program for the original specification as the source of truth. Random inputs were generated and given to both the original and simplified programs. The outputs between the programs were then checked for equality

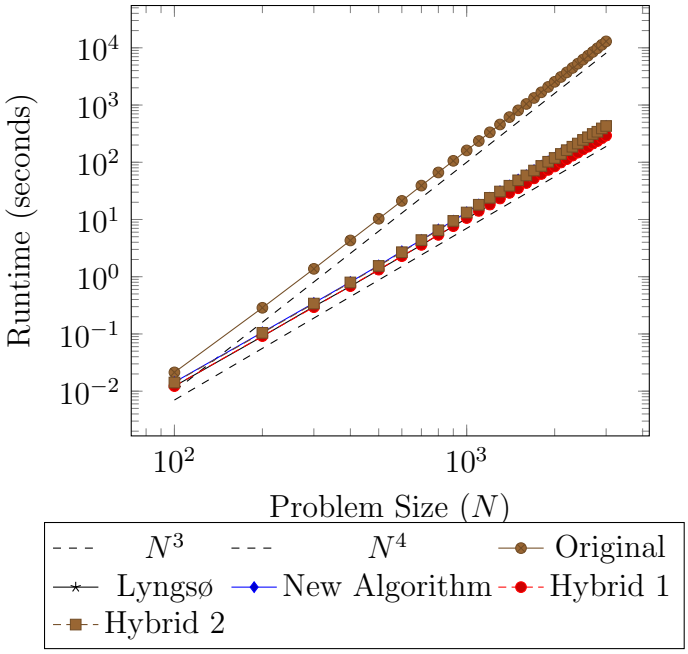


Figure 4.3: Average runtimes of all generated fast-i-loops programs, plotted alongside $(7\text{e}{-9})N^3$ and $(1\text{e}{-10})N^4$ to verify the polynomial degree of the runtimes with respect to the problem size N.

(within floating point precision). All of the simplified programs were found to produce the correct results for a variety of problem sizes over multiple executions.

All programs were executed on a Linux system equipped with an Intel Core i7-12700K CPU. The amount of memory used during each execution was confirmed to be less than the amount of memory available via the Linux `time` utility[1]. This ensured that memory paging did not affect performance.

Figure 4.3 shows a plot for the algorithm runtimes across problem sizes from $N = 100$ to $N = 3000$ in increments of 100. This plot uses logarithmic scaling for both axes, which causes polynomials to be shown as straight lines. Functions for $O(N^3)$, and $O(N^4)$ are plotted alongside the results to visually confirm that the measured complexities match what we theoretically expected. That is, the original specification runs in $O(N^4)$ time, while the simplified versions run in $O(N^3)$ time.

Based on the polynomials for the number of loop iterations per program (see Table 4.2),

[1] `https://www.man7.org/linux/man-pages/man1/time.1.html`

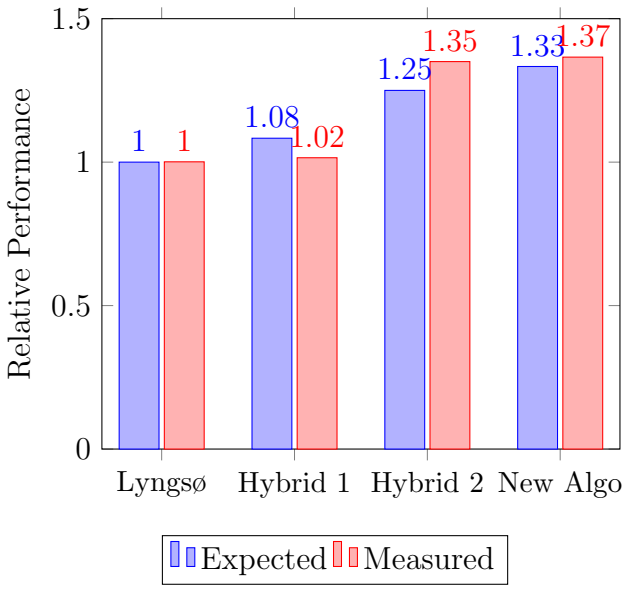


Figure 4.4: Relative performance of the simplified fast-i-loops programs averaged across all problem sizes. Expected values are computed from the leading term in the polynomial for the number of loop iterations. Measured values are the ratio of each program's runtime to the fastest program's runtime.

we expect the Lyngsø-equivalent program to have the best performance, followed closely by the "Hybrid 1" program. The "Hybrid 2" and "New Algorithm" programs are expected to be about 25% and 33% slower respectively. To evaluate the accuracy of this, Figure 4.4 reports the measured performance of each program relative to the fastest alongside the expected ratios. The greatest error in this estimation was for the "Hybrid 2" algorithm, where our measured performance differed from the expected performance by 7.72%. This shows that, for the tested programs, the execution time is an acceptable proxy for the total number of operations performed.

4.6 Future Work

Our compiler works only on *independent* reductions, i.e., those where the expressions/values over which the reduction operator is applied do not use *any instance* of the result variable. However, Yang et al. [89] show that dealing with *dependent* reductions, where some outputs may depend recursively on computed results, is an important problem in practice. Our

current implementation may produce equations that do not admit a legal schedule. This can, of course, be detected by running a scheduler on the simplified program. We are working on integrating scheduling with simplification in the standard recursion down the face lattice.

Consistent with Gautam and Rajopadhye [49], we only handle programs with a single size parameter. Extending the theory to multiple size parameters is important for many algorithms, e.g., in RNA computations, we may have two sequences of respective lengths, N and M. As noted by Loechner and Wilde [37], such domains may be decomposed into *chambers*, each with a distinct face lattice, hence polyhedral tools are capable of handling multiple parameters. However, the function describing complexity is still a multivariate polynomial involving two non-comparable parameters, N and M. This introduces a partial order among the complexity functions and requires extending the core simplification algorithm.

4.7 Conclusion

Nearly two decades after the theory was first proposed by Gautam and Rajopadhye [49], we implemented the reduction simplification transformation, a powerful program transformation capable of automatically exploiting reuse in programs involving reductions. The original theory omitted several key components required for employing simplification in practice, which we discussed and addressed, thus providing the first complete push-button implementation of simplification in a compiler.

We evaluated its effectiveness in automatically rediscovering several key results in algorithmic improvement previously only attainable through manual human analysis. In particular, we illustrated how simplification discovered three new cubic algorithms for RNA secondary structure prediction. Our work takes a step toward raising the level of abstraction for the user and demonstrates how what used to require clever, painstaking analysis can be systematically employed as a sequence of program transformations in a compiler.

Chapter 5

Maximal Simplification of Polyhedral Reductions

As we discussed in the previous chapter, when properly exploited, the simplification transformation of Gautam and Rajopadhye [49] enables an automatic improvement in the asymptotic complexity of programs containing reductions. As we have discussed, Gautam and Rajopad-

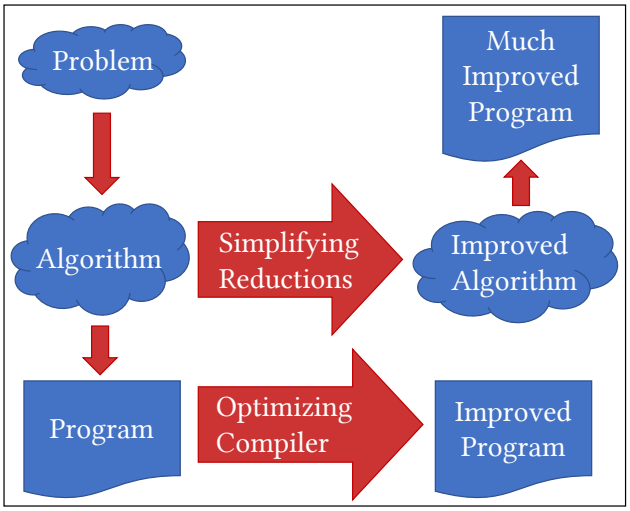


Figure 5.1: Simplification improves the asymptotic complexity of the *algorithm*.

hye is optimal, but not *maximal*, in the sense that it cannot handle all possible scenarios of reuse. The simplification algorithm may fail when the reduction operator does not admit an inverse. Such operators are very common in many dynamic programming algorithms. Indeed, polyadic dynamic programming problems [8] are nothing but reductions and are widely used in many bio-informatics algorithms [38, 3, 4, 47, 40, 65, 59, 63, 2, 50].

In this chapter, we propose a method to extend the simplification algorithm to handle all scenarios, achieving maximal simplification. At the heart of our approach is *piece-wise simplification*, the notion that we can split the problem into a number of pieces and then independently simplify each piece. Such *piece-wise affine transformations*, also known as index-set splitting, have a long history in other polyhedral analyses [28, 41, 71, 56]. The

difficulty lies in the fact that, in general, there are infinitely many ways to split a polyhedron. We give constructive proofs showing how to select a finite number of pieces for simplification. In doing so, we make the following contributions,

1. We propose extensions to the simplification algorithm to support any arbitrary independent reduction, particularly when the operator does not admit an inverse.

2. We provide an implementation of our approach in the AlphaZ system.

The remainder of this chapter is organized as follows. Section 5.1 provides several motivating examples with progressively increasing difficulty. In Sections 5.2 and 5.3, we formulate our approach and justify it in Sections 5.4 and 5.5. We discuss aspects of our implementation in Section 5.6. Finally, we review related work in Section 5.7 and conclude in Section 5.8.

5.1 Motivating Examples

We now show several simple examples to illustrate the intuition of simplification and its limitations.

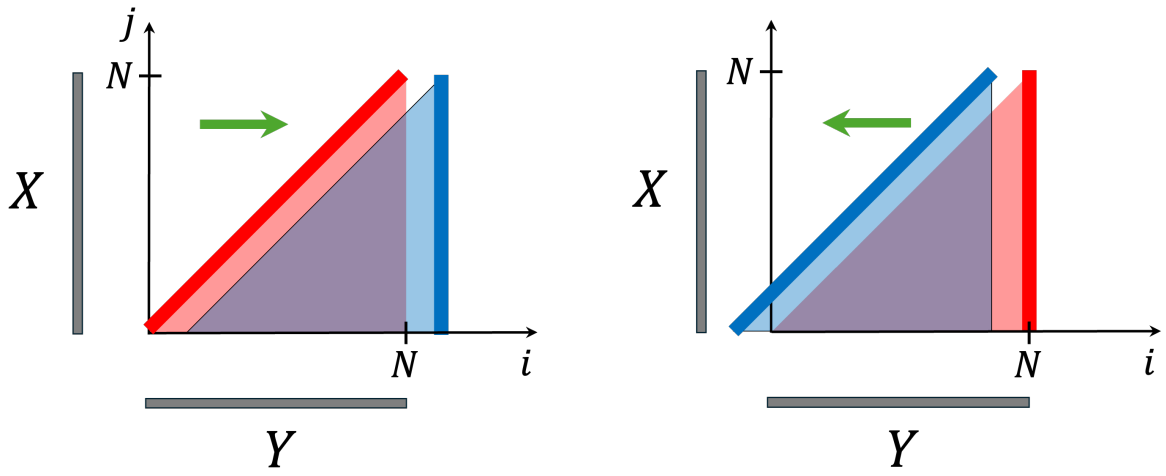
5.1.1 Prefix sum

Consider the prefix sum, which computes an N-element output array from an N-element input array where the i'th element of the output is the sum of the first i elements of the input:

$$Y_i = \sum_{j=0}^{j \leq i} X_j \tag{5.1}$$

As written, Equation 5.1 has an asymptotic complexity of $O(N^2)$ because each of the N elements in the output involves a summation over $O(N)$ input values. However, two adjacent elements in the output involve summing many of the *same elements* (i.e., there is *reuse*), which means there is an opportunity to reuse partial results.

For example, $Y_k = X_0 + X_1 + ... + X_k$ and $Y_{k+1} = X_0 + X_1 + ... + X_k + X_{k+1}$. Once Y_k is computed, there is no need to re-sum the first k elements of X to compute Y_{k+1}. Instead,



(a) Equation 5.2 expressing Y_i in terms of Y_{i-1}.

(b) Equation 5.3 expressing Y_i in terms of Y_{i+1}.

Figure 5.2: Reduction from quadratic to linear asymptotic complexity. The input variable X is oriented along the vertical axis, and the output Y is oriented along the horizontal.

Y_{k+1} can be expressed in terms of Y_k and only the *new* values of X:

$$Y_{k+1} = (X_0 + X_1 + X_2 + ... + X_k) + X_{k+1}$$

$$Y_{k+1} = Y_k + X_{k+1}$$

This observation allows Equation 5.1 to be rewritten as either:

$$Y_i = \begin{cases} i = 0 & : \quad X_0 \\ i > 0 & : \quad Y_{i-1} + X_i \end{cases} \tag{5.2}$$

where Y_i is computed from Y_{i-1}, or equally validly, as:

$$Y_i = \begin{cases} i = N & : \quad \sum_{j=0}^{N} X_j \\ i < N & : \quad Y_{i+1} - X_{i+1} \end{cases} \tag{5.3}$$

where Y_i is computed from Y_{i+1} instead. Geometrically, Equations 5.2 and 5.3 involve computation on only some of the 1D edges of the domain as opposed to the original equation

which involves a computation over the 2D triangular domain. The terms are colored to match the corresponding edges in Figure 5.2. Consequently, both have a smaller, better asymptotic complexity of $O(N)$. The latter requires an initial summation of all N elements of X to produce Y_N, but the overall complexity is still linear because the remaining $N-1$ elements are each computed in constant time. The goal of Gautam and Rajopadhye's simplification is to explore all such possible rewrites and find the ones with the optimal asymptotic complexity in the presence of reuse. However, as illustrated by the next example, some rewrites may not be possible.

5.1.2 Prefix max

Consider the prefix max, which is identical to the previous example, except it uses the max operator instead of addition:

$$Y_i = \max_{j=0}^{j \leq i} X_j \tag{5.4}$$

Everything else is the same, the value produced at Y_i can be used to compute the next value at Y_{i+1}, allowing it to be rewritten with $O(N)$ complexity as:

$$Y_i = \begin{cases} i = 0 & : \quad X_0 \\ i > 0 & : \quad \max(Y_{i-1}, X_i) \end{cases} \tag{5.5}$$

exactly like Equation 5.2. However, the key difference here is that only one of the rewrites is possible. There is no way to express an equation analogous to Equation 5.3 because the max operator does not admit an inverse. This is not an issue here since at least one rewrite does not involve the inverse operation, and thus simplification can find it. However, in general, a rewrite may not exist that does not involve the inverse operation, and consequently, simplification may fail, as illustrated by the next example.

5.1.3 Sliding and increasing max filter

Consider the following equation, which computes an N-element output array from an N-element input array where the i'th element of the output is the max over the sliding, and increasing, window from i to $2i$ on the input:

$$Y_i = \max_{j=i}^{j \leq 2i} X_j \tag{5.6}$$

As written, this has an asymptotic complexity of $O(N^2)$. Like the previous examples, there is also reuse. Across two adjacent answers in the output, $O(N)$ of the same inputs are accumulated:

$$Y_k = \max(X_k, X_{k+1}, X_{k+2}, ..., X_{2k})$$
$$Y_{k+1} = \max(\quad X_{k+1}, X_{k+2}, ..., X_{2k}, X_{2k+1}, X_{2k+2})$$

But, neither Y_{k+1} can be expressed in terms of Y_k nor can Y_k be expressed in terms of Y_{k+1} because this would require *removing* the contributions of either X_k or X_{2k+1} and X_{2k+2} which is not possible because the max operator has no inverse.

However, it is still possible to rewrite Equation 5.6 with an asymptotic complexity of $O(N)$, but doing so is surprisingly challenging as it requires constructing a series of clever splits. We describe how our approach handles this scenario to obtain an $O(N)$ simplification in Section 5.5, but we encourage the reader to try to work out an $O(N)$ time program to implement Equation 5.6 to fully appreciate the difficulty for themselves.

5.1.4 Working example

Previously in Chapter 4, we walked through the simplification of a 2-dimensional reduction. In this chapter, we will use a higher dimensional reduction as our working example. This

is necessary because several of the considerations necessary for maximal simplification only have meaning in 3D and higher.

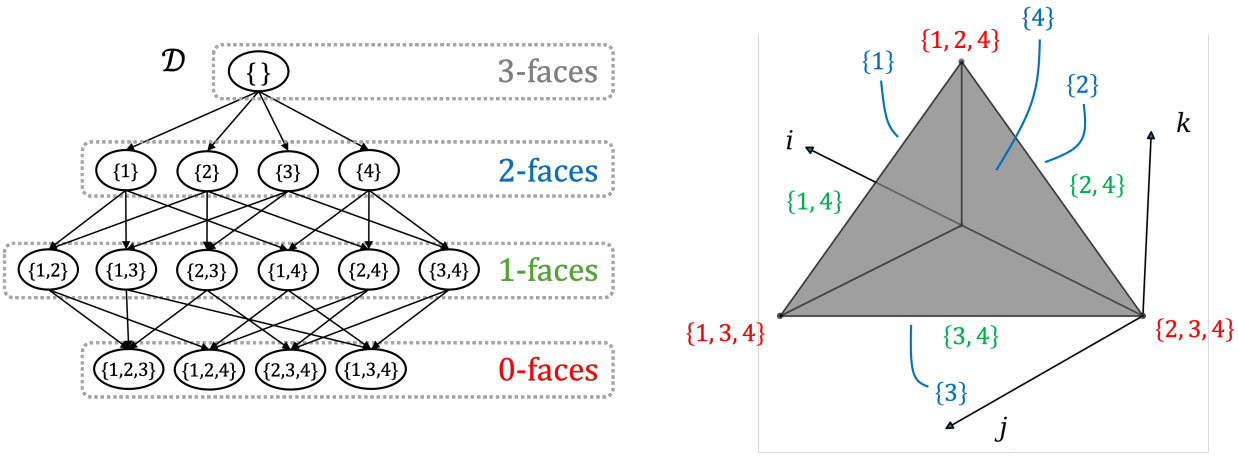


Figure 5.3: Face lattice of $\mathcal{D}$ in Equation 5.7. The numbers inside the labels denote the constraints that, when saturated, describe the face. For example, "{4}" denotes saturating constraint $c_4 : k \leq i - j$, which represents the top oblique 2-face. The back three edges (1-faces) and vertex (0-face) are not labeled.

Consider the reduction which produces an $O(N)$-element array specified by the following equation:

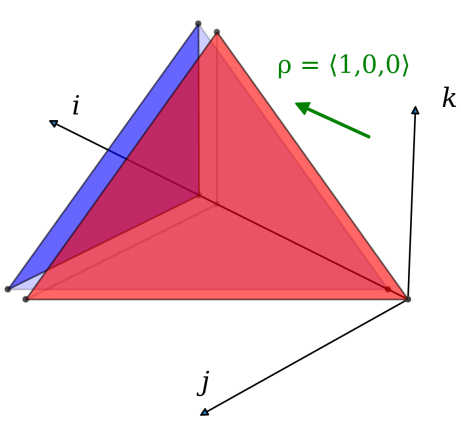
$$Y_i = \sum_{(i,j,k)\in\mathcal{D}} X_k \tag{5.7}$$

over the 3-dimensional domain:

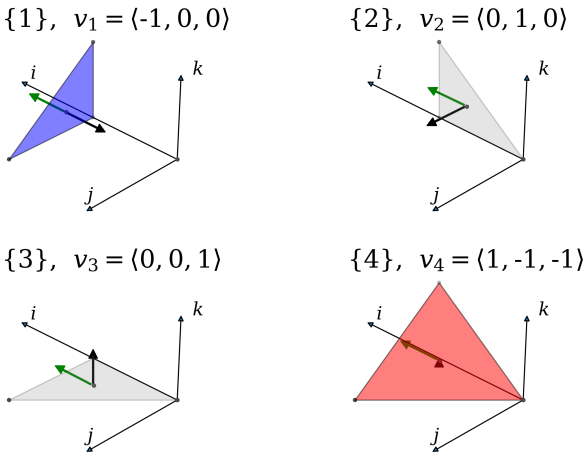
$$\mathcal{D} = \{[i,j,k] \mid (i <= N) \wedge (0 \leq j) \wedge (0 \leq k) \wedge (k \leq i - j)\} \tag{5.8}$$

where i, j, and k are indices and N is a parametric size parameter. The write function here is $f_p = \{[i,j,k] \to [i]\}$ and therefore the accumulation space is the 2-dimensional jk-plane, $\mathcal{A} = \{[i,j,k] \mid i = 0\}$. Similarly, the read function here is $f_d = \{[i,j,k] \to [k]\}$ and therefore the reuse space is the 2-dimensional ij-plane, $\mathcal{R} = \{[i,j,k] \mid k = 0\}$. This set has four inequality constraints: c_1 $(i \leq N)$, c_2 $(0 \leq j)$, c_3 $(0 \leq k)$, and c_4 $(k \leq i - j)$. Geometrically, the shape of $\mathcal{D}$ is a tetrahedron with 4 faces (2-faces), 6 edges (1-faces), and 4 vertices (0-faces). The face lattice of $\mathcal{D}$ is illustrated in Figure 5.3.

5.1.5 Single-Step Simplification



(a) Translate $\mathcal{D}$ (shift red pyramid) by $\vec{\rho}$ (green arrow) yielding $\mathcal{D}_s$ (blue pyramid).



(b) Evaluate the residual computation on only the facets (here, 2-faces) of $\mathcal{D}$ and/or $\mathcal{D}_s$.

Figure 5.4: Simplification of a cubic equation (computation defined over a tetrahedron) to a quadratic complexity (the residual computation is defined only over the points in the top red triangular face).

Our working example involves a reduction with the addition operator. The i'th result, Y_i, is the accumulation of the values of the reduction body at points in the i'th triangular slice of the tetrahedron. The complexity of the equation is the number of integer points in $\mathcal{D}$, namely $O(N^3)$. Simplification is possible here because the body has redundancy along any vector $\vec{\rho} \in \mathcal{R}$, such as $\vec{\rho} = \langle 1, 0, 0 \rangle$ (green arrow) shown in Figure 5.4. Any two points $[i, j, k], [i', j', k'] \in \mathcal{D}$ separated by a scalar multiple of $\vec{\rho}$ read the same value of X because $X_k = X_{k'}$. In other words, the body expression evaluates to the same value at all points along $\langle 1, 0, 0 \rangle$. Simplification exploits this reuse to read and compare only the $O(N^2)$ distinct values. Recall that the geometric explanation of simplification, illustrated in Figure 5.4, is to translate $\mathcal{D}$ (shift the red pyramid) by $\vec{\rho}$ (green arrow) yielding $\mathcal{D}_s$ (the blue pyramid). Then delete all computations in the intersection of the two and evaluate the residual computation on only (a subset of) the facets (here, 2-faces) of $\mathcal{D}$ and/or $\mathcal{D}_s$.

Additionally, some of these facets can be ignored. This is a ***critical*** component of how we will construct splits and is discussed further in Sections 5.2.2 and 5.2.3. For now, note

that the grey triangular 2-faces "{2}" and "{3}", whose normal vectors are orthogonal to $\vec{\rho}$, were already included in the intersection, and the back blue 2-face "{1}" one at $i = N+1$ is *external* since it does not contribute to any answer. This leaves a residual computation on only the top red oblique 2-face "{4}". Thus, the $O(N^3)$ computation in Equation 5.7 can be rewritten as the following $O(N^2)$ equation:

$$Y_i = \begin{cases} i = 0 \;\; : \;\; \sum_{j=0}^{j \leq i} X_{i-j} \\ i > 0 \;\; : \;\; Y_{i-1} + \sum_{j=0}^{j \leq i} X_{i-j} \end{cases} \tag{5.9}$$

obtained from the application of Theorem 5 from Gautam and Rajopadhye [49], which we do not review in detail here. The intuition is that simplifying Equation 5.7 along $\vec{\rho} = \langle 1, 0, 0 \rangle$ *moves* the computation to the "{4}" face, which is described by the saturated constraint $k = i - j$. Thus the subscript, k, on X from Equation 5.7 is expressed as X_{i-j} in Equation 5.9.

5.1.6 Equivalence Partitioning of Infinite Choices

In this example, the reuse space is multi-dimensional, which means there are infinitely many choices of reuse along which simplification can be performed. However, not all choices of reuse need to be explored. The intuition is that any two reuse vectors resulting in the same combination of residual computation can be viewed as the same candidate choice. In the previous section, the choice of $\vec{\rho} = \langle 1, 0, 0 \rangle$ resulted in a residual computation on only the top oblique 2-face. Gautam and Rajopadhye [49] refers to the subset of $\vec{\rho}$ that results in the same residual computation as an *equivalance class*. Then, the set of all equivalence classes can be explored with dynamic programming. This guarantees the final simplified program's optimality as long as a single reuse vector from each equivalence class is considered.

Let $\mathcal{F}$ be an arbitrary facet of the reduction body. Let $\mathcal{F}_i$ denote the i'th facet of $\mathcal{F}$ (i.e., its i'th child), and let $\vec{\nu_i}$ be the linear part of the normal vector of $\mathcal{F}_i$. Let the symbol $\oplus$ be the reduction operator, and $\ominus$ be its inverse if $\oplus$ is invertible. The single-step simplification

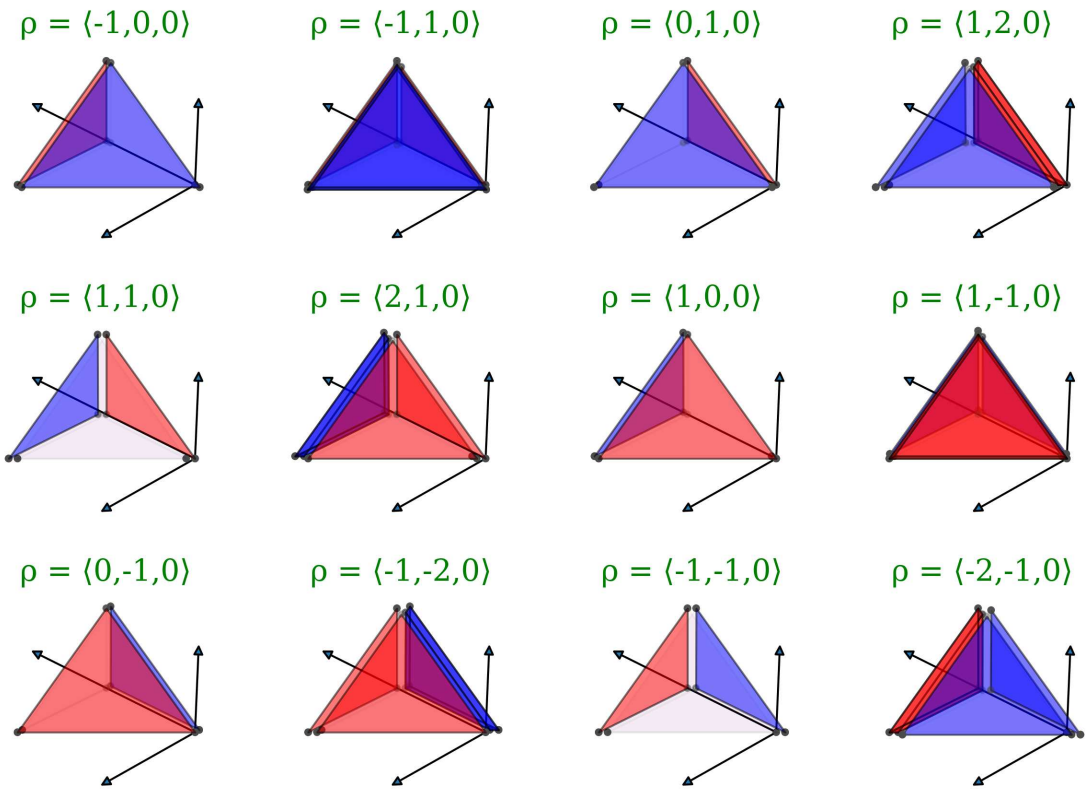


Figure 5.5: The 12 equivalence classes for the single-step simplification of Equation 5.7.

of $\mathcal{F}$, summarized in Section 5.1.5, with the reuse vector $\vec{\rho}$, results in a residual computation on some of its facets. The orientation of $\vec{\rho}$ relative to each facet dictates the type of residual computation that occurs on $\mathcal{F}_i$. There are three possibilities, depending on the sign of the dot product between $\vec{\rho}$ and $\vec{\nu_i}$:

1. If $\vec{\rho} \cdot \vec{\nu_i} > 0$ then $\mathcal{F}_i$ contributes with the $\oplus$ operator.

2. If $\vec{\rho} \cdot \vec{\nu_i} < 0$ then $\mathcal{F}_i$ contributes with the $\ominus$ operator.

3. If $\vec{\rho} \cdot \vec{\nu_i} = 0$ then $\mathcal{F}_i$ does not contribute at all.

Each facet $\mathcal{F}_i$ may be labeled as either an $\oplus$-face, $\ominus$-face, or $\oslash$-face respectively. [1] We say that $\vec{\rho}$ *induces a particular labeling*, $\mathcal{L}$, on the facets of $\mathcal{F}$. Each labeling corresponds to one of the equivalence classes. For a face with m facets, there are a total of 3^m distinct labelings.

[1] The notion of *labelings* are not part of the prior work of Gautam and Rajopadhye [49], we introduce this language to help explain why simplification can fail in the following sections.

However, many of these will not be possible. There is no way to mark all facets with the same label, for example, since the set of $\vec{\rho}$ with a positive dot product to all normal vectors in a convex polytope is empty. In this example, the 12 possible labelings (equivalence classes) are shown in Figure 5.5 with $\oplus$-faces colored in red, $\ominus$-faces in blue, and $\oslash$-faces uncolored.

5.1.7 Recursive Simplification

In the general case, reductions have a d-dimensional reduction body, an a-dimensional accumulation, and an r-dimensional reuse space. The process of Section 5.1.5 is applied recursively on the face lattice, starting with $\mathcal{D}$. At each step, we simplify the facets of the current face $\mathcal{F}$. The key idea is that exploiting reuse along $\vec{\rho}$ avoids evaluating the reduction expression at most points in $\mathcal{F}$. Specifically, let $\mathcal{F}'$ be the translation of $\mathcal{F}$ along $\vec{\rho}$. Then all the computation in $\mathcal{F} \cap \mathcal{F}'$ is avoided, and we only need to consider the two differences $\mathcal{F}' \backslash \mathcal{F}$ and $\mathcal{F} \backslash \mathcal{F}'$, i.e., the union of some of the facets of $\mathcal{F}$.

At each recursive step down the face lattice, the asymptotic complexity is reduced by exactly one polynomial degree, as facets of $\mathcal{F}$ are strictly smaller dimensional subspaces. Furthermore, at each step, the newly chosen $\vec{\rho}$ is linearly independent of the previously chosen ones. Hence, the method is optimal—all available reuse is fully exploited. This holds regardless of the choice of $\vec{\rho}$ at any level of the recursion, even though there may be infinitely many choices.

Bringing everything together, the residual $O(N^2)$ computation in Equation 5.9 of our working example:

$$Y_i = \sum_{j=0}^{j \leq i} X_{i-j} \tag{5.10}$$

can be thought of as a completely new 2-dimensional reduction that may be further simplified. We do not describe this in detail here, as it is very similar to the prefix sum described in Section 5.1.1 and we have already walked through a 2D example previously in Chapter 4.

5.2 Splitting to avoid simplification failure

In this section, we present the intuition of our main result with a simple example. The subsequent sections provide the precise formulation and proofs for the general cases.

5.2.1 How and when simplification can fail

As discussed in the previous section, the simplification algorithm proceeds recursively down the face lattice, simplifying facets one by one while additional reuse exists. At each step of the recursion, on a particular face $\mathcal{F}$, several candidate choices of reuse are explored, one for each possible labeling of the facets of $\mathcal{F}$. For a particular labeling, each facet is treated as either an $\oplus$-face, $\ominus$-face, or $\oslash$-face. Let us assume now that the reduction operator does not admit an inverse, which means that $\ominus$-faces must be avoided. Recall that some facets may be ignored (e.g., all but the top red oblique 2-face in Figure 5.4), but some can not; let us refer to these facets that can *not* be ignored as *residual facets*. For a particular labeling, simplification is impossible when two residual facets exist with opposite labels, one $\oplus$-face, and one $\ominus$-face. If all residual facets are $\oplus$-faces, then simplification can obviously proceed. Similarly, if they are all $\ominus$-faces, we can simply negate $\vec{\rho}$ to view them as $\oplus$-faces. The single-step simplification of Section 5.1.5 fails if all candidate choices of reuse induce labelings that involve residual facets with opposite labels.

An example

Look back at the prefix max from Section 5.1.2. The reduction body is the triangle, $\mathcal{D} = \{[i, j] \mid 0 \leq j \leq i \leq N\}$ and the reuse space is the 1D space along the i-axis, $\mathcal{R} = \{[i, j] \mid j = 0\}$. This means that there are only two labelings, shown on the left of Figure 5.6. The labeling induced by $\vec{\rho} = \langle 1, 0\rangle$ involves a single residual $\oplus$-face. In contrast, the labeling induced by $\langle -1, 0\rangle$ involves two residual facets with opposite labels (i.e., one red and one blue coloring). Since at least one labeling exists with no oppositely labeled residual facets, simplification succeeds and enables us to write Equation 5.5. The vertical patterned blue

facet on the left triangle can be ignored because it is a boundary facet as discussed in Section 5.2.2.

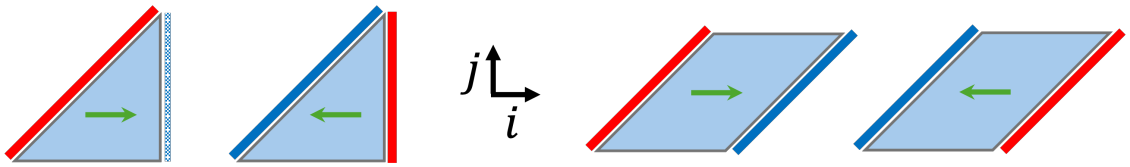


Figure 5.6: The two labelings for the prefix max example from Section 5.1.2 shown by the triangles on the left. The two labelings for the parallelogram in Equation 5.11 are shown on the right. Residual facets are colored in solid red ($\oplus$-faces) and blue ($\ominus$-faces). The vertical patterned blue facet on the left triangle is an outward boundary facet. Therefore, it can be ignored, and thus, the labeling induced by $\langle 1, 0 \rangle$ is possible since this does not involve residual facets with opposite labels.

Now consider the following example:

$$Y_i = \max_{\substack{(j \leq i) \wedge (j \leq N) \\ (0 \leq j) \wedge (i - N \leq j)}} X_j \tag{5.11}$$

This reduction has a body in the shape of a parallelogram with two possible labelings, as shown on the right of Figure 5.6. Since all (two) labelings involve at least one pair of residual facets with opposite labels, simplification fails.

Avoid labelings with oppositely labeled residual facets

Given this formulation characterizing when simplification fails, our goal is to split the reduction body into pieces so that we can guarantee the existence of at least one labeling in each piece that involves only residual $\oplus$-faces exclusively or $\ominus$-faces, but not both. Before we can describe how to do this, we need to augment Gautam and Rajopadhye's [49] original characterization of non-residual facets, i.e., the facets that may be ignored, which we will call *boundary* and *invariant* facets.

5.2.2 Boundary facets

Let $\mathcal{H}$ denote the linear space of the facet $\mathcal{F}$, defined by Gautam and Rajopadhye as the intersection of the effectively saturated constraints characterizing $\mathcal{F}$. An unsaturated con-

straint, c_i in $\mathcal{F}$ is characterized as a *boundary* constraint if the following condition holds:

$$\mathcal{H} \cap \mathcal{A} \subseteq \mathcal{H} \cap \ker(c_i) \tag{5.12}$$

where $\mathcal{A}$ is the accumulation space defined previously, and $\ker(c_i)$ is the null space of the linear part of the constraint. This says that a facet is a boundary if its linear space contains the entire accumulation space, i.e., multiple points *on that face* contribute to one or more answers. Boundary facets are useful because any boundary facet labeled as an $\ominus$-face can be ignored since the "answer(s)" are not needed.

We will make an additional distinction on the degree to which a facet is considered a boundary. Let us further characterize boundary facets as *strong* or *weak* based on the following definitions.

Definition 1. *Let a facet of $\mathcal{F}$, defined by the effectively saturated constraint c_i, be called a* **strong** *boundary facet if the following condition holds,*

$$\operatorname{rank}\big(\mathcal{A} \cap \ker(c_i)\big) = \operatorname{rank}\big(\mathcal{A}\big) \tag{5.13}$$

Definition 2. *Let a facet of $\mathcal{F}$, defined by the effectively saturated constraint c_i, be called a* **weak** *boundary facet if the following condition holds,*

$$0 < \operatorname{rank}\big(\mathcal{A} \cap \ker(c_i)\big) < \operatorname{rank}\big(\mathcal{A}\big) \tag{5.14}$$

In other words, a strong boundary facet is one where no other point contributes to the answer(s) to which the points on the facet contribute. In contrast, a facet whose linear space contains *part of* (i.e., has a non-trivial and incomplete intersection with) the accumulation space is said to be a weak boundary. Note that these are mutually exclusive; a facet can not be simultaneously strong and weak.

An example

The distinction between strong and weak boundaries only has meaning in 3D and higher. Look back at the working example from Section 5.1.4. The accumulation space in this example is the jk-plane. Of its four 2-faces, shown in Figure 5.4, the "{1}" face, at $i = N$, is a strong boundary because its linear space, the jk-plane contains the entire accumulation space. The other three 2-faces are all weak boundaries, because the intersection of their linear spaces with the accumulation space is a 1D subspace.

5.2.3 Invariant facets

Gautam and Rajopadhye [49] does not explicitly characterize what we will call invariant facets. We can think of an invariant facet as the dual of a boundary facet, but from the perspective of the reuse space instead of the accumulation space. Let an unsaturated constraint c_i in $\mathcal{F}$ be characterized as an *invariant* constraint if the following condition holds:

$$\mathcal{H} \cap \mathcal{R} \subseteq \mathcal{H} \cap \ker(c_i) \tag{5.15}$$

where $\mathcal{R}$ is the reuse space. Invariant facets are useful because the recursion never proceeds into an invariant facet (i.e., invariant facets are always labeled as $\oslash$-faces regardless of the choice of reuse $\vec{\rho}$).

Similarly, we distinguish the extent to which a facet is invariant based on the following definitions.

Definition 3. *Let a facet of $\mathcal{F}$, defined by the effectively saturated constraint c_i, be called a* **strong** *invariant facet if the following condition holds,*

$$\operatorname{rank}\big(\mathcal{R} \cap \ker(c_i)\big) = \operatorname{rank}\big(\mathcal{R}\big) \tag{5.16}$$

Definition 4. *Let a facet of $\mathcal{F}$, defined by the effectively saturated constraint c_i, be called a*

weak *invariant facet if the following condition holds,*

$$0 < \text{rank}\big(\mathcal{R} \cap \ker(c_i)\big) < \text{rank}\big(\mathcal{R}\big) \tag{5.17}$$

Note that a facet may be simultaneously a weak invariant and a weak boundary. This happens when the intersection of accumulation space, reuse space, and linear space of the facet is non-trivial.

An example

Again, looking back at the working example in Section 5.1.4, the reuse space is the ij-plane. The bottom "{2}" face is,, therefore,, a strong invariant facet, and the other faces are weak invariant facets. In other words, any vector $\vec{\rho} = \langle \rho_i, \rho_j, 0 \rangle \in \mathcal{R}$ labels the bottom 2-face an $\oslash$-face because any such $\vec{\rho}$ is orthogonal to the normal vector of the bottom 2-face, $\langle 0, 0, 1 \rangle$.

5.2.4 Residual facets

As mentioned in Section 5.2.1, some facets may be ignored during single-step simplification. Let us refer to the facets that can not be ignored as residual facets, which are defined as follows:

Definition 5. *A facet of $\mathcal{F}$, defined by the effectively saturated constraint c_i, will be called a* ***residual*** *facet if the following condition holds,*

$$\Big(\text{rank}\big(\mathcal{A} \cap \ker(c_i)\big) < \text{rank}\big(\mathcal{A}\big)\Big) \vee \Big(\text{rank}\big(\mathcal{R} \cap \ker(c_i)\big) < \text{rank}\big(\mathcal{R}\big)\Big) \tag{5.18}$$

In other words, residual facets are neither strong boundary facets nor strong invariant facets. Residual facets are those into which the recursion may need to explore that can potentially be labeled as $\ominus$-faces.

5.2.5 Split Reduction

Le Verge [27] showed that a reduction in the form of Equation 3.2, with the body $\mathcal{D}$ that can be partitioned into disjoint subsets $\mathcal{D}_1$ and $\mathcal{D}_2$ (i.e., $\mathcal{D} = \mathcal{D}_1 \cup \mathcal{D}_2$ and $\mathcal{D}_1 \cap \mathcal{D}_2 = \phi$) may be rewritten as the following:

$$Y_{f_p(z)} = \begin{cases} f_p(\mathcal{D}_1) \cap f_p(\mathcal{D}_2) & : \left(\bigoplus_{z \in \mathcal{D}_1} X_{f_d(z)} \right) \oplus \left(\bigoplus_{z \in \mathcal{D}_2} X_{f_d(z)} \right) \\ f_p(\mathcal{D}_1) \setminus f_p(\mathcal{D}_2) & : \bigoplus_{z \in \mathcal{D}_1} X_{f_d(z)} \\ f_p(\mathcal{D}_2) \setminus f_p(\mathcal{D}_1) & : \bigoplus_{z \in \mathcal{D}_2} X_{f_d(z)} \end{cases} \tag{5.19}$$

Even though the domain is only split into *two* pieces, there may be *three* branches because the pieces may overlap in the answer space.

An example

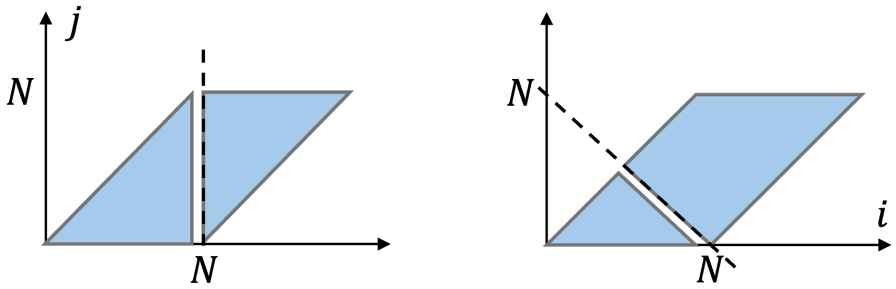


Figure 5.7: Two example splits of the reduction body of Equation 5.11, the split along $i = N$ shown on the left and the split along $i + j = N$ on the right. Both cases have two pieces, but the split by $i + j = N$ results in three branches of the transformed equation because the pieces overlap when projected onto the answer space (i.e., the i-axis).

The reduction body in the example from Equation 5.11 is the parallelogram, $\mathcal{D} = \{[i, j] \mid (0 \leq j \leq N) \wedge (i - N \leq j \leq i)\}$. Consider one split of $\mathcal{D}$ by the hyperplane $i = N$, where $\mathcal{D}_1 = \mathcal{D} \cap \{[i, j] \mid i < N\}$ and $\mathcal{D}_2 = \mathcal{D} \cap \{[i, j] \mid i \geq N\}$ shown on the left of Figure 5.7.

Spliting the reduction in Equation 5.11 along $i = N$ yields:

$$Y_i = \begin{cases} 0 \leq i < N & : \max\limits_{j=0}^{j \leq i} X_j \\ N \leq i \leq 2N & : \max\limits_{j=i}^{j \leq N} X_j \end{cases} \tag{5.20}$$

Equation 5.20 only has two branches because the projection of the pieces onto the answer space do not overlap (i.e., project the triangles on the left of Figure 5.7 down onto the i-axis).

Consider another split $\mathcal{D}$ by the hyperplane $i+j = N$, where $\mathcal{D}_1 = \mathcal{D} \cap \{[i,j] \mid i+j < N\}$ and $\mathcal{D}_2 = \mathcal{D} \cap \{[i,j] \mid i+j \geq N\}$ shown on the right of Figure 5.7, yielding:

$$Y_i = \begin{cases} (N \leq 2i) \wedge (i < N) & : \max\left(\left(\max\limits_{j \in \mathcal{D}_1} X_j\right), \left(\max\limits_{j \in \mathcal{D}_2} X_j\right)\right) \\ 0 \leq 2i < N & : \max\limits_{j \in \mathcal{D}_1} X_j \\ N \leq i \leq 2N & : \max\limits_{j \in \mathcal{D}_2} X_j \end{cases} \tag{5.21}$$

In this case, there are three branches because the triangle and quadrilateral on the right of Figure 5.7 overlap from $N/2 \leq i < N$ when projected onto the i-axis.

5.2.6 Strong Boundary and Invariant Splits

In general, there are infinitely many ways to split the reduction body, $\mathcal{D}$, but not all are useful. Recall that simplification can fail when the reduction operator does not admit an inverse because a pair of oppositely labeled residual facets may be unavoidable. Splitting isolates some of the facets of $\mathcal{D}$ into one piece, making it possible to separate pairs of conflicting facets. Each piece retains some facets of $\mathcal{D}$ and obtains a single new facet characterized by the splitting hyperplane. We want to split $\mathcal{D}$ so that each piece's reuse space is less constrained, thereby enabling simplification.

However, an arbitrary split on $\mathcal{D}$ may undesirably further constrain the available reuse space of each piece because the newly introduced facet changes the set of new labelings

permissible in each piece. However, we can avoid this issue by only making splits that introduce a new strong boundary or strong invariant facet. This is useful because strong boundary $\ominus$-faces and invariant facets are never residual and can therefore be ignored.

If $\mathcal{D}$ is d-dimensional, then let h be a $(d-1)$-dimensional hyperplane that separates $\mathcal{D}$ into two non-empty pieces. The hyperplane h is characterized by a single equality constraint, c_h. Let $\ker(c_h)$ denote the null space of the linear part of c_h.

Definition 6. *Let h be called a strong boundary split if $\mathcal{A} \subseteq \ker(c_h)$.*

Definition 7. *Let h be called a strong invariant split if $\mathcal{R} \subseteq \ker(c_h)$.*

For example, the vertical split at $i = N$ shown previously in Figure 5.7 illustrates a strong boundary split because the accumulation space of the reduction in Equation 5.11 is $\mathcal{A} = \{[i, j] \mid i = 0\}$, which is indeed a subset of the null space of the linear part of $i = N$ (they happen to be the same space in this example). Consequently, the resultant reductions in each branch of Equation 5.20 can be simplified independently because there exists a labeling involving only residual $\oplus$-faces for each. These are indeed both instances of the prefix max discussed in Section 5.1.2.

5.3 Problem formulation and hypotheses

As we have discussed, simplification is a powerful transformation that lowers the asymptotic complexity of the underlying computation. However, it is not always possible as motivated by the reduction examples in Equations 5.6 and 5.11. In this section, we make our primary claim in the form of Theorem 1. Like Gautam and Rajopadhye [49], we assume that the input program only involves a single size parameter and any reductions present are *independent* (i.e., there is no cycle in the dependence graph among the variables appearing inside the reduction body and the answer variable on the left-hand side of the container equation).

Theorem 1. *Given an independent reduction with a d-dimensional body, an a-dimensional accumulation space, and an r-dimensional reuse space, it may always be transformed into an*

equivalent reduction with an asymptotic complexity that has been decreased by $l = \min(a, r)$ *polynomial degrees.*

The proof of Theorem 1 will follow from Sections 5.4 and 5.5 where we show how to handle all possible combinations of the dimensionalities of the accumulation space, reuse space, and reduction body that can occur. For each case, we will show that the reduction can be split into pieces such that each piece has at least one possible labeling without any residual $\ominus$-faces.

5.3.1 Assumptions

We make several assumptions about the input reductions to justify Theorem 1. We emphasize that this does not introduce any loss of generality and the following assumptions are made solely to facilitate the proofs in the following sections.

Separate Accumulation and Reuse Dimensions Only

We must only consider reductions involving separate accumulation and reuse dimensions, where $a + r = d$. Any reduction where $a + r \neq d$ can be systematically transformed into one or more instances of reductions where $a + r = d$. Therefore, it is sufficient to only consider reductions with accumulation and reuse dimensions.

First, consider the case where $a+r > d$. In such cases, the accumulation and reuse spaces have a non-trivial intersection, which means that the reduction accumulates the *same value* at many points in the body (i.e., along this intersection). Gautam and Rajopadhye [49] describes special simplification cases that may be applied to remove this intersection, which they call *higher order operator* and *idempotence* simplifications. The working example from Section 5.1.4 is in an instance of this case; the reduction body is 3-dimensional while the accumulation and reuse spaces are 2-dimensional, though we did not employ these special simplifications.

Second, consider the case where $a + r < d$. Such cases can be viewed as families of reductions, with $d-(a+r)$ *independent parameters*, reading independent slices of the inputs

and producing independent slices of the outputs. For example, the reduction:

$$Y[i,j] = \max_{k=i}^{2i} X[j,k]$$

has a 3D domain, 1D accumulation, and 1D reuse space. However, the index j should be viewed as an independent parameter. Thus, the overall computation can be viewed as $O(N)$ instances of independent 2D reductions, each with a 1D accumulation and 1D reuse space, one for each value of j, embedded in a 3D space. No simplification is possible among the different instances (each of the $O(N)$ reduction instances along j must be computed), but further simplification may be possible within the ik dimensions).

Orthogonal Accumulation and Reuse Along Canonical Axes

Let us assume that the accumulation and reuse spaces are *orthogonal* and are oriented along the canonical axes. Let the reuse space be along the first r canonical axes and the accumulation space be along the last a axes. The program variable domains and the indexing expressions can always be reindexed to put the accumulation and reuse along the canonic axes as described by Le Verge [27].

Domain of the Reduction Body is a Simplex

We restrict our analysis to domains that are simplexes (i.e., hyper-triangular), based on the following definition, adapted from Gruber [54].

Definition 8. *A (d)-simplex is the (d)-dimensional polytope defined as the convex combination of* $d+1$ *affinely independent vertices.*

In practice, decomposing the domain into simplices may not always be necessary. However, we use properties of simplices to prove that simplification is always possible. There may be heuristic solutions to decide how to split non-simplices, but we do not consider any such approaches here. Therefore, in the remaining discussion, we assume that any reductions appearing in the input program have been preprocessed and initially decomposed into

simplices.

Our maximal simplification result directly carries over to general polyhedral sets because any (d)-dimensional parametric polytope can be decomposed into the union of (d)-dimensional simplices. Triangulating multi-dimensional polytopes is common in computer graphics [5, 34], for example. Simplices have useful properties used in our proofs:

- Any (k)-face of an (d)-simplex is itself a (k)-simplex. The number of (k)-faces of a (d)-simplex are given by the binomial coefficients, $\binom{d+1}{k+1}$ unique combinations of its $d+1$ vertices.

- A (d)-simplex can be split into two (d)-simplices by adding one new affinely independent vertex. See Lemma 1.

5.3.2 Simplex-Preserving Strong Boundary or Invariant Splits

Repeatedly trying to make arbitrary splits runs the risk of falling into an endless loop. We need a way to guarantee that the process of splitting will terminate. Recall that simplification of a reduction with the body $\mathcal{D}$ fails when every possible labeling involves one or more pairs of oppositely labeled *residual* facets. Therefore, if we can repeatedly split $\mathcal{D}$ in such a way that *both* preserves the total number of facets in each piece *and* strictly decreases the number of residual facets, then it will be possible to guarantee that each piece has a labeling with no conflicting residual facets.

We combine the following two ideas. First, we will use the fact that splitting a d-simplex through any of its $(d-2)$-faces produces two d-simplices per Lemma 1 and therefore preserves the total number of facets in each piece. Second, we will use the notion of a strong boundary or invariant split which introduces a single new *non-residual* facet, as described in Section 5.2.6. Combining these two ideas, by making strong boundary or invariant splits through $(d-2)$-faces of simplices, guarantees that the process of splitting will produce pieces with strictly fewer residual facets because the number of total facets in each piece remains the same. Such splits will be called Simplex-Preserving strong Boundary (SPB) or Invariant

(SPI) splits.

When the accumulation space is one-dimensional, an SPB split can be constructed by infinitely extending one of the $(d-2)$ faces along it. Similarly, when the reuse space is 1-dimensional, extending a $(d-2)$-face along it yields an SPI split. Since there are finitely many $(d-2)$-faces, there are finitely many candidate SPB and SPI splits to process.

Lemma 1. *Let a splitting hyperplane of a (d)-dimensional polytope be any (d−1)-dimensional hyperplane that has points on both sides of the hyperplane. Any splitting hyperplane that saturates a (d-2)-face of a (d)-simplex produces two (d)-simplices.*

Proof. By definition, an (d)-simplex is the convex combination of $(d+1)$ vertices. Additionally, every (k)-face is itself a (k)-simplex which is just the simplex formed by the $(k+1)$ vertices of the (k)-face. Any splitting $(d-1)$-dimensional hyperplane that saturates a $(d-2)$-face contains its $(d-1)$ vertices. There are two remaining vertices and these by definition can not be part of the splitting hyperplane. This is because the hyperplane is itself a $(d-1)$-simplex and if it contained one of these additional two vertices, then it would saturate an entire $(d-1)$-face of the domain. Therefore, we can consider these other two vertices as separate from the hyperplane. We will refer to these as vertex A and vertex B. The convex combination of vertices A and B forms a (1)-simplex (i.e., a (1)-face or 1-dimensional linear subspace that connects the vertices). Let point C, be any point in this (1)-simplex. We can construct the following two new sets.

1. The convex combination of the $(d-1)$ vertices on the $(d-2)$-face, vertex A, and point C.

2. The convex combination of the $(d-1)$ vertices on the $(d-2)$-face, vertex B, and point C. Then take the set difference of this set with the previous set.

Since both are convex combinations of $(d+1)$ vertices, they are both, by definition, (d)-simplices. □

A 2D example

The vertical split at $i = N$ illustrated previously in Figure 5.7 is analogous to an SPB split, in the sense that is a strong boundary split through a vertex. However, a parallelogram is not a simplex of course. Regardless, we can compute the constraint $i = N$ as follows. First compute the linear space, H, of the bottom right vertex of the parallelogram in Figure 5.7, as the intersection of the saturated constraints describing this vertex:

$$H = \{[i, j] \mid (j = 0) \wedge (j = i - N)\}$$

Next construct the relation characterizing the 1-dimensional basis, b, of the accumulation space, from the kernel of the write function, $f_p = \{[i, j] \rightarrow [i]\}$ in this example, to obtain the relation:

$$b = \{[i, j] \rightarrow [i, j + 1]\}$$

Then compute its transitive closure, b^*, to obtain the relation:

$$b^* = \{[i, j] \rightarrow [i, j']\}$$

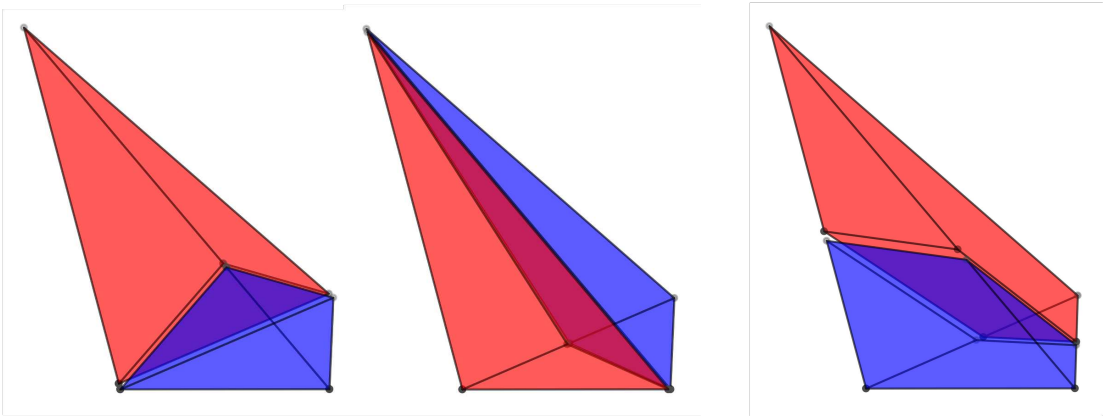
Then apply b^* to the linear space of the vertex, H, to obtain the set:

$$\{[i, j] \mid i = N\}$$

Finally use the single equality constraint to characterize the splitting hyperplane $i = N$. Extending H by b^* effectively removes one of the equality constraints, because extending it increases its dimensionality by one. Since we started from a set with two equality constraints (i.e., because H came from a $(d-2)$-face) its extension is guaranteed to have a single equality constraint. These operations are available in the integer set library, `isl` [62].

A 3D example

In three dimensions, a simplex-preserving split is any plane passing through an edge of a tetrahedron. Any such plane splits it into two tetrahedra (one of which may be empty) as illustrated in Figure 5.8.



(a) Two boundary splits, each through an edge. (b) Arbitrary split.

Figure 5.8: *Tetrahedron*-preserving strong boundary splits (shown on left). Axes are not shown, but imagine that the accumulation space is 1D along the k-dimension pointing upwards. This means each split here is a strong boundary split per Section 5.2.6, but if the split does not pass through an edge then the resulting pieces are not guaranteed to be tetrahedra, as shown on the right. Splits passing through edges, however, result in two tetrahedra as shown on the left.

5.4 3-dimensional and Higher Reductions

In this section, we consider the single-step simplification of Section 5.1.5 for 3-dimensional and higher reductions where all possible labelings involve at least one pair of oppositely labeled facets. Therefore, splitting the reduction body may be required. Our goal is to show that pairs of oppositely labeled residual facets can always be avoided. We will show that all but three residual facets can *either* be transformed into strong boundary facets by reduction decomposition *or* ignored by choosing a reuse vector that labels them as $\oslash$-faces. Two of the remaining three residual facets may involve opposite labels. Whenever this happens, we will split the domain so that all $\oplus$-faces are on one side of the split and all $\ominus$-faces are on the other. Then in the piece containing the $\ominus$-face(s), we will negate $\vec{\rho}$ to view them as $\oplus$-faces.

We rely on the fact that any $(d-1)$-face has a non-trivial intersection with subsets of the accumulation and reuse space oriented along the canonical axes, per Lemma 2.

Lemma 2. *The linear space of any $(d-1)$-face has a non-trivial intersection with the linear subspace spanned by any two or more canonic axes.*

Proof. The $(d-1)$-dimensional linear space describing any $(d-1)$-face involves d indices and one equality constraint. The linear subspace spanned by q canonic axes involves d indices and $(d-q)$ equality constraints and is q-dimensional. Their intersection involves $(1+d-q)$ equality constraints. Therefore it is $(q-1)$-dimensional. When $q > 1$, this intersection is non-trivial. □

5.4.1 Avoid Some Residual Facets with Reduction Decomposition

The reduction decomposition transformation described previously in Section 4.4.2 is useful because it has the effect of transforming a weak boundary into a strong boundary. This happens because the accumulation space of the inner reduction (i.e., the null space of f'_p in Equation 4.12) is strictly smaller after decomposition. Recall that a weak boundary, per Definition 5.2.2, is a facet that partially intersects the accumulation space. By constructing the inner accumulation space to only involve the dimensions contained by a weak facet, the facet becomes a strong boundary in the inner reduction. Multiple weak boundary facets can simultaneously be transformed into strong boundary facets by constructing the inner accumulation as the subset of the accumulation space that they share.

Lemma 3. *Given a dimensions of accumulation, any set of a residual facets can be transformed into $a-1$ strong boundary facets via reduction decomposition.*

Proof. Let $[\mathcal{F}_1, \mathcal{F}_2, ..., \mathcal{F}_k]$ be a list of k residual facets. Take the first residual facet, $\mathcal{F}_1$, and let $\mathcal{A}_1$ be the intersection of its linear subspace $\mathcal{H}_1$ and the a-dimensional accumulation space $\mathcal{A}$,

$$\mathcal{A}_1 = \mathcal{H}_1 \cap \mathcal{A} \tag{5.22}$$

$\mathcal{A}_1$ is a $(a-1)$-dimensional subspace per Lemma 2. Now let $\mathcal{A}_{p-1}$ denote the intersection of the first $(p-1)$ residual faces and the accumulation space,

$$\mathcal{A}_{p-1} = \mathcal{H}_1 \cap \mathcal{H}_2 \cap ... \cap \mathcal{H}_{p-1} \cap \mathcal{A} \tag{5.23}$$

$\mathcal{A}_{p-1}$ is a $(a-p+1)$-dimensional subspace. Now we decompose the reduction, per Section 4.4.2, where the inner reduction's accumulation space is precisely along $\mathcal{A}_{p-1}$. This is done for up to a residual facets since $\mathcal{A}_{p-1}$ is at least 1-dimensional when $p = a$. The inner reduction now has a 1D accumulation space, and $p-1$ of the residual facets are subsequently strong boundaries (i.e., non-residual). □

5.4.2 Avoid Residual Facets with Appropriate Reuse Selection

Any facets labeled as an ⊘-face can be ignored. Multiple facets can be labeled as an ⊘-face by selecting a reuse vector orthogonal the all of the facet normal vectors. The following proof is analogous to that of Theorem 3.

Lemma 4. *Given r dimensions of reuse, any set of $r-1$ residual facets can be labeled as ⊘-faces by choosing a reuse vector in their combined intersection with the reuse space.*

Proof. Let $[\mathcal{F}_1, \mathcal{F}_2, ..., \mathcal{F}_k]$ be a list of k residual facets. Take the first facet $\mathcal{F}_1$ and let $\mathcal{R}_1$ be the intersection of its linear space $\mathcal{H}_1$ with the r-dimensional reuse space $\mathcal{R}$,

$$\mathcal{R}_1 = \mathcal{H}_1 \cap \mathcal{R} \tag{5.24}$$

$\mathcal{F}_1$ is an $(r-1)$-dimension subspace. Let $\mathcal{R}_{p-1}$ denote the intersection of the first $(p-1)$ residual facets and the reuse space,

$$\mathcal{R}_{p-1} = \mathcal{H}_1 \cap \mathcal{H}_2 \cap ... \cap \mathcal{H}_{p-1} \cap \mathcal{R} \tag{5.25}$$

$\mathcal{R}_{p-1}$ is a $(r-p+1)$-dimensional subspace. Any reuse vector $\vec{\rho} \in \mathcal{R}_{p-1}$ is orthogonal to the

normal vectors of all $p-1$ facets and therefore labels all as $\oslash$-faces.

□

5.4.3 All possible scenarios

Now consider an arbitrary reduction over a d-dimensional simplicial domain with a dimensions of accumulation and r dimensions of reuse. There are $d+1$ facets on the reduction body because it is a simplex. We just showed, in the previous subsections, how to avoid up to $(a-1)+(r-1) = d-2$ of them using reduction decomposition and an appropriate choice of reuse. Consequently, there are only two possible scenarios that must be considered.

Base Case: $d-1$ or fewer residual facets

In this case, $a-1$ residual facets can be transformed into strong boundaries per Lemma 3 and $r-1$ facets can be labeled as $\oslash$-faces per Lemma 4. This leaves at most one remaining residual facet because $(d-1)-(a-1)-(r-1) \leq 1$. At least two residual facets must be present for simplification to fail; therefore, simplification succeeds in this case.

General Case: d or more residual faces

Like the base case, $a-1$ residual facets can be transformed into strong boundaries and $r-1$ facets can be labeled $\oslash$-faces. In this case, however, there can be up to three remaining residual facets since $(d+1)-(a-1)-(r-1) \leq 3$. Among the remaining residual facets, two of them may involve opposite labels. In other words, there may be one $\oplus$-face and two $\ominus$-faces, or two $\oplus$-faces and one $\ominus$-face. In this case, a single SPB or SPI split can be made through one of their $(d-2)$-faces to separate the conflicting facets.

5.5 2-Dimensional Reductions: Fractal Simplification of Triangles

Only one type of reduction can occur in two dimensions: a reduction with a 1D accumulation and 1D reuse space. Per Section 5.3.1, let the reuse space be oriented horizontally along the

i-axis and accumulation vertically along the j-axis. This means that vertical and horizontal edges are boundary and invariant edges.

There are only three types of triangles that can occur:

1. 1 residual edge (i.e., a right triangle, which is an instance of a standard scan, e.g., Section 5.1.2)

2. 2 residual edges (some of which require fractal simplification)

3. 3 residual edges, (can be split into two disjoint instances each with 2 residual edges)

The following sections describe how to make 2-dimensional versions of SBP or SBI splits, previously described in Section 5.3.2, to obtain instances of the first case.

5.5.1 Base Case: Right Triangles

Any triangle with only one residual edge can always be simplified. If the residual edge monotonically increases, we exploit reuse along $\vec{\rho} = \langle 1, 0 \rangle$. This yields a standard scan (e.g., prefix-max, prefix-min, etc.). Otherwise, we exploit reuse along $\langle -1, 0 \rangle$, producing a backward scan (suffix-max, suffix-min, etc.)

5.5.2 Two Residual Edges

In this case there is either one boundary (vertical) or one invariant (horizontal) edge. Let the three vertices of the triangle be, $\{\mathcal{V}_0, \mathcal{V}_1, \mathcal{V}_2\}$, of which, $\mathcal{V}_0$, which we call the corner vertex, is the intersection of the two residual edges. There are two sub-cases depending on the relative orientation of the vertices.

Covered corner

Let the corner vertex (the vertex between the two residual edges) be called *covered* if it lands between the other two vertices when all vertices are projected onto either of the axes. As illustrated by the red point in Figure 5.9, and by the definition of $\mathcal{V}_0$ being covered, its projection on the other edge is a point on that edge. Hence, an SPB or SPI split through

$\mathcal{V}_0$ yields two right triangles. One can be simplified as a forward-scan and the other as a backward-scan.

Corner vertex is not covered

There is no loss of generality in assuming that the entire triangle is in the positive quadrant and the corner vertex is at the origin (this can be accomplished by a simple reindexing of one or both of the input/output arrays). Let us first consider that the non-residual edge is vertical, and let $\mathcal{V}_1$ be *below* $\mathcal{V}_2$. If n is sufficiently large, we make one horizontal cut through the lower vertex, $\mathcal{V}_1$, and let $\mathcal{V}_2'$ be its intersection with the upper residual edge. Next, we make a vertical cut through $\mathcal{V}_2'$, intersecting the lower residual edge at, say, $\mathcal{V}_1'$. We now have three triangles, $\Delta_1 = [\mathcal{V}_0, \mathcal{V}_1', \mathcal{V}_2']$, $\Delta_2 = [\mathcal{V}_1', \mathcal{V}_2', \mathcal{V}_1]$ and $\Delta_3 = [\mathcal{V}_2', \mathcal{V}_1, \mathcal{V}_2]$. Of these, the latter two are right triangles and Δ_1 is geometrically similar to the original triangle as illustrated on the left side of Figure 5.10.

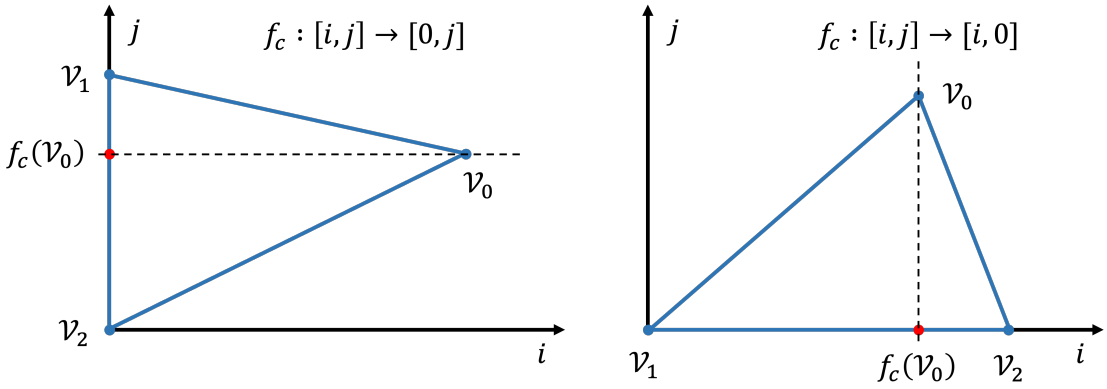


Figure 5.9: Covered corner vertex: The projection of $\mathcal{V}_0$ (shown by the red point) is between $\mathcal{V}_1$ and$\mathcal{V}_2$. A single horizontal or vertical cut through the corner produces two right triangles.

Theorem 2. *The reduction over* $[\mathcal{V}_0, \mathcal{V}_1, \mathcal{V}_2]$ *can be simplified, i.e., computed with* $O(1)$ *reduction operations per element of the answer variable* Y*.*

Proof. By Induction. As the base case, consider $n \leq c$, for some constant c. For such sufficiently small triangles, computing each element of Y needs only $O(1)$ reduction operations. In general, when n is sufficiently large, let us assume by induction that Δ_1 can be simplified

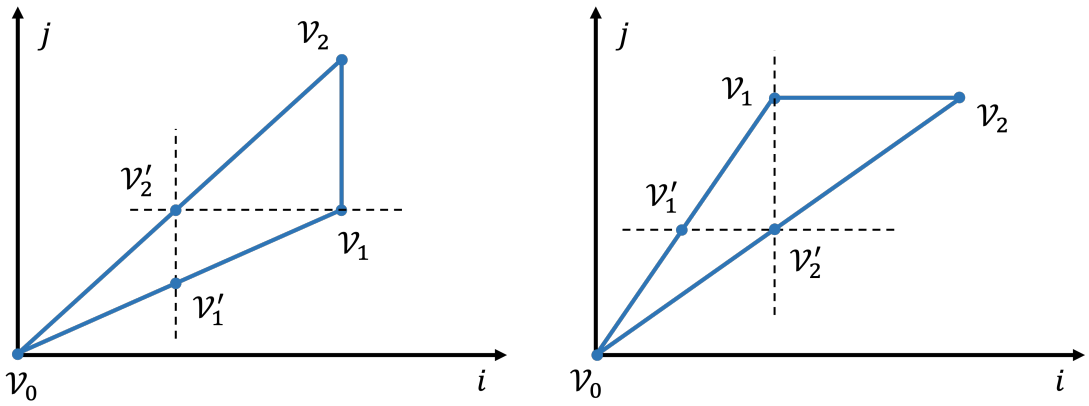


Figure 5.10: Non-covered corner vertex: $\mathcal{V}_0$ lands *outside* of the other two vertices when projected onto the linear space of the vertical edge (left) or when projected onto the space of the invariant edge (right). Each case requires a single vertical and horizontal cut to produce two good right triangles in the base case and a third triangle which is a homothetic scaling of the original triangle.

to compute the section of the result until $\mathcal{V}'_1$ in $O(1)$ time per element of Y. Another backward scan, and a forward scan on appropriate section of X, will yield the next section of Y from $\mathcal{V}'_1$ to $\mathcal{V}_1$. □

The above argument can be carried over mutatis mutandis to the case where the non-residual edge is horizontal. Indeed, as illustrated on the right side of Figure 5.10, the first cut of the previous case yields the second one.

The proof of Theorem 2 suggests a simple code generation strategy (again, explained for the case when the non-residual edge is vertical). Note that vertices $\mathcal{V}_1$ and $\mathcal{V}_2$ are given as affine functions of the parameter, n, and this is known statically. Let Let $\mathcal{V}_1 = [an, bn]$ and $\mathcal{V}_2 = [an, b'n]$ for some scalars a, b and b'. We can compute the values of a_1, b_1 and b'_1 the factors that specify $\mathcal{V}'_1$ and $\mathcal{V}'_2$, and from that, the scaling factor, $\frac{a}{a_0} = \frac{b}{b'_0} = \frac{b'}{b'_0}$. This leads to the code structure shown in Figure 5.11. It is important to note that this code is recursive, not polyhedral, and takes us out of the polyhedral model. Nevertheless, all our analyses are polyhedral, and in order to generate it, we need only a fixed number of splits. Also, note that the recursive function calls can be optimized using standard tail recursion optimization techniques.

```
void fractal(int *Y, int *X, int L, int U) {
    if (U < threshold) {                          // do the full input reduction
        for (i=L; i<U; i++)
            for (j=i; j<=2*j; j++)
                Y[i] = max(Y[i], X[j]);
        return;
    }
    for (i=U; i>=U/2; i--)                        // backward scan on U/2<=i<=U
        Y[i] = max(Y[i+1], X[i]);
    for (i=U/2; i<=U; i++)                        // foward scan on U/2<=i<=U
        Y[i] = max(Y[i-1], X[2*i], X[2*i-1]);
    fractal(Y, X, L, U/2);                        // recurse on L<=i<U/2
}
```

Figure 5.11: Recursive pseudo-code for fractal simplification of the example from Section 5.1.3.

Case 2: three residual edges

In this case any vertical cut through a covered vertex when projected onto the i-axis will produce two triangles in case 2. Alternatively, and equivalently, any horizontal cut through a covered vertex when projected on the j-axis will achieve the same thing.

5.6 Implementation

We provide a proof-of-concept implementation of the individual components of our approach using the Alpha language [14, 21] and the AlphaZ system [70]. The Alpha language is an equational language that separates the specification of a program from its execution plan. Additionally, it supports modeling reduction operations as first-class objects with explicit representations of the write and read functions, f_p and f_d, characterizing the accumulation and reuse space, respectively. Program variable domains are represented as polyhedral sets using `isl` [62] (the integer set library), which naturally supports the constraint representations we use to describe our splitting hyperplanes.

Our results in this paper are primarily theoretical. A complete push-button tool that implements the extended dynamic programming algorithm and automatically and optimally simplifies any reduction requires addressing several practical issues. Specifically, it is neces-

sary to handle two issues: managing the combinatorially large number of possible solutions, and developing methods to compare them using the constant factors, and the existence of parallel schedules with scalable locality. Putting all this together is outside the scope of this paper, and left as future work.

This can be used to produce simplified Alpha programs for all of the examples discussed above. Additionally, AlphaZ can be used to generate C code that can be subsequently compiled and run.

5.7 Related Work

Simplification has garnered renewed interest recently. Asymptotic inefficiencies are present, even in deployed codes. Ding and Shen [77] noted that nine of the 30 benchmarks in Polybench 3.0, and two deployed PDE solvers have such inefficiencies. Separately, Yang, et al. [89] showed that simplification is useful for many algorithms in statistical learning like Gibbs Sampling (GS), Metropolis Hasting (MH) and Likelihood Weighting (LW). Their bencharks include Gaussian Mixture Models (GMM), Latent Dirichlet Allocation (LDA) and Dirichlet Multinomial Mixtures (DMM). See their paper for details of benchmarks, algorithms, size parameters, machine specs, etc.

Simplification is closely related to tensor contractions, a problem that arises in many domains. They can be computed optimally using Pearl's "summary passing" or "message passing" algorithm [12] for Bayesian inference, or the generalized distributive law (GDL) due to Aji and McEliece [39]. GDL has been discovered in many different contexts and is equivalent to the sum-product algorithm proposed by Kschischang et al. [43] and its general framework described by Shafer and Shenoy [19]. We first explain the technique through a simple example drawn from Aji and McEliece [39] and then describe how it complements reduction simplification. Consider the following system of equations:

$$X_{i,l} = \sum_{j,k=1}^{N} A_{i,j,l} \times B_{i,k} \qquad\qquad Y_j = \sum_{i,k,j=1}^{N} A_{i,j,l} \times B_{i,k} \tag{5.26}$$

Naively, each equation would have $O(N^4)$ complexity since each result is the accumulation of a triple summation, and there are $O(N)$ answers. However, if we define two new variables:

$$T_{i,l} = \sum_{j=1}^{N} A_{i,j,l} \qquad\qquad T'_i = \sum_{k=1}^{N} B_{i,k} \tag{5.27}$$

then the following is an equivalent simplified system of equations, and the complexity is only $O(N^3)$:

$$X_{i,l} = T_{i,l} \times T'_i \qquad\qquad Y_j = \sum_{i,l=1}^{N} A_{i,j,l} \times T'_i \tag{5.28}$$

The GDL algorithm efficiently evaluates such systems of equations, in particular, those that formulate a problem called the *marginalization of product functions* (the MPF problem), which is nothing but a set finite number of tensor contractions, using a set of common input tensors. A large number of problems in diverse domains can be formulated as MPFs and therefore the GDL serves a unifying, generic algorithm leading to cross-disciplinary insight. GDL consists of building what is called a junction tree or forest from the sets of labels representing the input and output variables. The construction of this forest is guaranteed to succeed, possibly at the expense of additional auxiliary variables, and the computation of the functions is essentially a bottom-up traversal of the forest (possibly followed by a top-down one for multiple outputs).

The GDL and related algorithms have recently had a big impact, especially in the design and re-emergence of low-density parity checking codes, and have provided deep insight into the links between belief propagation in artificial intelligence and coding theory. In certain instances, new algorithms have been discovered using this framework [45, 53]. Nevertheless, GDL has usually been applied by hand: we are not aware of tools that implement it automatically. A typical scientist seeking to use these methods tends to (i) write the equations; (ii) study their algebraic and structural properties; (iii) build the underlying graph struc-

ture (in the case of GDL); (iv) develop insight into the intermediate equations to define; (v) write down the optimized equations; and (vi) eventually produce an executable program that embodies the new algorithm.

MPF problems, or tensor contractions, are a subset of the class of the equations we handle via simplification. Typically the algorithms that optimize them handle multiple contractions, all involving the same set of input tensors. The complementarity of the polyhedral model and GDL is highlighted by the following observations:

In tensor contractions, the reuse space is only along a subset of the canonic vectors: no "oblique" reuse is allowed. This is because the dependences on the right-hand side are only a *permutation of a subset* of the indices. Note that in some instances (e.g., belief propagation) it may not even make sense to formulate an equation with oblique reuse. Furthermore, the problem where some oblique reuse is present has received some attention [46], but the optimizations only exploit distributivity and not simplification.

Similarly, the accumulation space is along a subset of the canonic vectors, and no "oblique" accumulations are allowed. Again, this may not make sense in some domains. Moreover, most *initial* specifications for simplification method also do not have oblique accumulations. However, some equations require an oblique decomposition of the accumulation space to optimally expose/exploit the reuse. Because of this, the reuse and accumulation spaces can be represented as a subset of index names, rather than arbitrary subspaces of the index space (note that the power set of index names is finite, albeit combinatorially large, but there are infinitely many vector subspaces). There is a distinct subset for each subexpression in the product expression.

The domains of the GDL accumulation expressions are *hyper-rectangular parallelepipeds*. Again, no "oblique" boundaries are allowed. Hence, scans cannot be expressed and/or detected in the MPF formulation.

Finally, in many tensor contractions, the complexity is measured in terms of the *number* of subexpressions, and the bounds of the summations are assumed to be (small) constants,

leading to "exponential" complexity. Simplification on the other hand, comes from the domain of loops and compiler optimization, so the accumulation bounds are program size parameters, and the number of subexpressions are small constants, and the program complexity is polynomial in the parameters.

5.8 Conclusions and Open Questions

The ultimate goal is to enable compilers to take a high-level application program specification and carry it out in the most efficient way possible, preferably *automatically* and *optimally*. This work takes a step in that direction to enable users (i.e., application scientists and programmers) to focus less on the *engineering* aspects (the *how*) of their algorithms and more on the problems (the *what*) that their algorithms are intended to solve. In this work, we have studied reuse-based simplification of polyhedral reductions. The simplification transformation proceeds recursively down the face lattice of the domain of the reduction body and attempts to exploit one dimension of available reuse at each level. Previously, at some point along this traversal, it may not have been possible to employ the simplification transformation without requiring inverse operations. However, as we have shown, it is always possible to split the residual reduction at the problematic node in the lattice into a finite number of simplicial pieces in such a way that we can guarantee that the simplification transformation will not fail. We have provided the mathematical proofs which enable us to make this claim.

We showed how to maximally simplify any arbitrary independent commutative reduction to obtain asymptotic the optimal asymptotic complexity. We evaluated a proof-of-concept implementation of the individual components of our approach that can be used to generate code. Along this same lines, the additional work concerning traditional compile-time scheduling and efficient code generation is still needed. For example, we rely on the fact that any polyhedral set can be decomposed into a union of disjoint simplices, but we do not discuss the implications of the choice of decomposition on the efficiency of the resulting code. At this point, for example, it is not obvious why we should prefer one decomposition

over another. It may be the case that one particular simplicial decomposition scheme leads to code that is more amenable to vectorization. These are interesting questions that require future exploration.

Chapter 6

Automatic Algorithm-Based Fault Tolerance of Stencil Computations

Computing technology continues to move in the direction of *more* capability and *more* complexity in *less* space with *less* power. Transient silent errors that originate in the hardware and manifest as silent data corruption pose a serious concern for software reliability. Such errors occur due to phenomena such as cosmic radiation [67, 84] and hardware component aging and degradation [85, 55]. This is a problem of both size and scale. At one end of the spectrum, as trends push the development of smaller and lower power systems, the likelihood of encountering such errors increases [60, 68]. On the other end, silent data corruption errors have even been observed in large-scale infrastructure services running at the global scale as recently as 2021 [87]. This also has implications in the realm of High-Performance Computing (HPC) [76, 72] where workloads can run for weeks and even months. Waiting months for the computation to finish only to realize that the result was incorrect has a significant impact given the time wasted. The need for robust fault tolerance today is becoming more and more prevalent as computing platforms grow in capability and complexity.

One approach to enable fault tolerance involves duplication either in the software [78, 48] or in the hardware [52]. Duplication has the highest coverage, but also the highest overhead. Other approaches avoid duplication and employ compile-time analysis to augment the software with checksums to detect errors in the memory [73]. While this is less expensive, it also has lower coverage. Silent errors that happen elsewhere, inside the floating-point arithmetic units as just one example, go undetected. Detection of silent transient errors is difficult because the errors manifest *in the data* and doing so requires running some analysis *on the computed data*, which most hardware-based fault tolerance schemes ignore.

Algorithm-based fault tolerance (ABFT) [7] has been widely studied since it was first

proposed in 1984 and provides a relatively cheap way to detect, and correct, such errors *in the data*. The main idea is to augment the computation with extra work in the form of invariant checksums by exploiting algebraic identities, which by definition should remain constant valued as the program executes. By comparing checksums evaluated periodically as the program executes, we can detect errors if their difference is above some threshold. ABFT schemes have been studied on a variety of different computational kernels from dense linear algebra such as Fast-Fourier Transform networks, matrix multiplication, and more recently convolutional neural networks and stencil computations [66, 58, 32, 90, 86, 36, 82].

In this work, we focus on ABFT in the context of scientific computing. Specifically, we focus on stencil applications which are commonly used to compute approximate solutions to partial differential equations modeling physical phenomena such as (electro)dynamic wave propagation and heat flow. Even among "just" stencil computations, there is a wide range of variability. The structure of a particular stencil depends on the specific properties of the particular phenomenon under study, such as the rate at which heat flows through (potentially multiple and different) physical media, just to give an example. These flow rates may be the same in all physical directions, referred to as *isotropic* diffusion in the literature, or direction-dependent, *anisotropic* [51]. To complicate things even further, these rates need not be the same everywhere, they may have different *magnitudes* at different physical points in space. All of these factors influence the particular form of the input stencil program, which directly affects the analysis required to carry out ABFT.

There are several challenges that make the application of algorithm-based fault tolerance difficult:

1. identification of the ABFT-applicable regions of the input program (and there may be multiple independent regions)

2. construction of the invariant checksums

3. computing the checksums efficiently enough to be worth it (i.e., with low overhead)

Each of these is largely dependent on the input program and requires very careful and manual analysis. However, there is no such *ABFT-compiler* (yet) that can perform all of the above from the input program alone. In this work, we propose a new methodology to carry out this analysis at compile time, embodied as a sequence of program transformations, which can be automated thanks to polyhedral compilation techniques. To this end, we make the following contributions:

- We show how to carry out an ABFT analysis automatically in a compiler.
- We present preliminary data on the tradeoff between overhead and error detection effectiveness of the generated code.

The rest of this paper is organized as follows. In Section 6.1, we provide several motivating examples showing how ABFT works on stencils illustrating the challenges mentioned above. The framework and scope of our analysis are reviewed in Section 6.2 followed by the description of our compiler passes in Section 6.3 and preliminary performance data and a discussion with respect to execution time (i.e., overhead) and effectiveness (i.e., ability to reliably detect errors) in Sections 6.4. Finally, we discuss related work and open questions in Sections 6.5 and 6.6.

6.1 Motivating Examples

Algorithm-based fault tolerance works fundamentally by defining invariant checksums over subsets of the domains of the computed variables and then asserting that the checksums do not change as the program evolves. For each checksum (in general, there may be multiple) this involves constructing two algebraically equivalent expressions that compute the same value, each taken from different program slices, and then asserting that their difference is zero[1].

[1]In reality, we deal with floating point arithmetic which is not associative so the difference will never truly be zero. Instead, the common practice is to assert that it is below some sufficiently small threshold.

Stencils are typically implemented as a series of weighted convolutions. The properties mentioned above manifest in the code as different weight values and potentially different convolution kernels at different points in the domain. This directly influences where it is possible to construct the invariant checksum expressions. We illustrate this variability with several examples of increasing complexity in the following sections. One quickly appreciates the difficulty of doing this manually and can see why it is desirable to leave such analysis to a polyhedral compiler.

6.1.1 1D Jacobi stencil with constant weights

Consider the Jacobi 1D stencil which updates an $(N+1)$-element array over a series of T time steps. The primary computation of each non-boundary point comes from the weighted sum of three neighboring points from the previous time step.

$$B_{t,i} = \begin{cases} A_i & \text{if } \ t = 0 \\ B_{t-1,i} & \text{elif } \ 0 < t \leq T \text{ and } i = 0 \text{ or } i = N \\ w_0 B_{t-1,i-1} + w_1 B_{t-1,i} + w_2 B_{t-1,i+1} & \text{else} \end{cases} \tag{6.1}$$

In stencils, deriving the invariant checksum is achieved by building two expressions such that no two points with the same time step (i.e., the t index value of $B_{t,i}$ in Equation 6.1) appear in both expressions.

Construct invariant checksums

Let us introduce a new variable, $C_{t,l,m}$, to denote the checksum at time step t defined as the sum over the window of values of $B_{t,i}$ for $l \leq i \leq m$,

$$C_{t,l,m} = \sum_{i=l}^{m} B_{t,i} \qquad 1 \leq l \leq m \leq N-1 \tag{6.2}$$

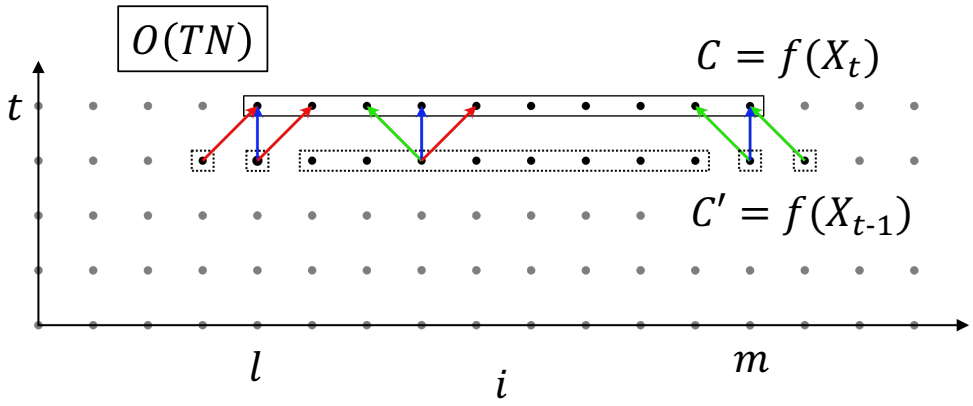


Figure 6.1: Single checksum illustration. Points contributing to $C_{t,l,m}$ in Equation 6.2 shown in the solid boxed region and points contributing to $C'_{t,l,m}$ in Equation 6.4 in dashed boxed regions. Repeating this each time step at every $(m-l)$'th position along i leads to an overall cost of $O(TN)$.

This is illustrated as the solid boxed region in Figure 6.1. By substituting the definition of $B_{t,i}$ where $1 \leq l \leq m \leq N-1$ from Equation 6.1 into Equation 6.2 we obtain,

$$C_{t,l,m} = \sum_{i=l}^{m} \left(w_0 B_{t-1,i-1} + w_1 B_{t-1,i} + w_2 B_{t-1,i+1} \right) \tag{6.3}$$

Notice that the points of B, in Equation 6.3, can be grouped based on the combination of w_0, w_1, and w_2 through which they contribute, shown by the dashed boxes in Figure 6.1. The middle dashed rectangle represents the points at time step $t-1$ that contribute to C_t by all three weights. After some algebra we obtain,

$$\begin{aligned} C'_{t,l,m} \equiv\ & (w_0) B_{t\text{-}1,l\text{-}1} + (w_0 + w_1) B_{t\text{-}1,l} \\ & + (w_0 + w_1 + w_2) \sum_{i=l+1}^{m-1} B_{t\text{-}1,i} \\ & + (w_1 + w_2) B_{t\text{-}1,m} + (w_2) B_{t\text{-}1,m+1} \end{aligned} \tag{6.4}$$

which we will denote as $C'_{t,l,m}$. The right-hand sides of Equations 6.2 and 6.4 both compute the *same numerical value* using *different elements* in the domain of the variable B. Equation 6.2 uses elements solely from time step t, and Equation 6.4 from time step $t-1$.

The only way for $C_{t,l,m}$ and $C'_{t,l,m}$ to evaluate to different values is if some error were to occur between their computation. To use this, we assert that the absolute value of their difference,

$$\Delta C_{t,l,m} \equiv |C'_{t,l,m} - C_{t,l,m}|$$

is below some threshold, large enough to be distinguishable from floating-point round-off errors, and small enough to actually detect most errors. This is discussed further in Section 6.4.

Achieve low overhead with interpolation

For any fault tolerance scheme to be feasible, the cost of implementing it needs to be sufficiently low. However, computing and comparing checksums as shown in the previous section has a significant amount of overhead. This is because the work required to compute $\Delta C_{t,l,m}$ from Equation 6.2 in Figure 6.1 is asymptotically the same as the main stencil computation. The overhead here is too high, however, it is possible to do better.

Instead of comparing expressions across *adjacent* time steps where,

$$C' = f(B_{t-1,i}) \tag{6.5}$$

we could compare them across *several*, say τ, time steps such that C' is a function of only the points in B at time step $t - \tau$,

$$C' = f(B_{t-\tau,i}) \tag{6.6}$$

To achieve this, we can repeatedly substitute B in C'. After two substitutions we will have $C' = f(B_{t-2,i})$, after three substitutions $C' = f(B_{t-3,i})$, and so on as illustrated in Figure 6.2. At each step, we need to account for the combinations of weights. As we show in Section 6.3.1 since the checksum accumulates points computed by a convolution, this leads to a very regular pattern, but still, this is not something one would want to do by hand.

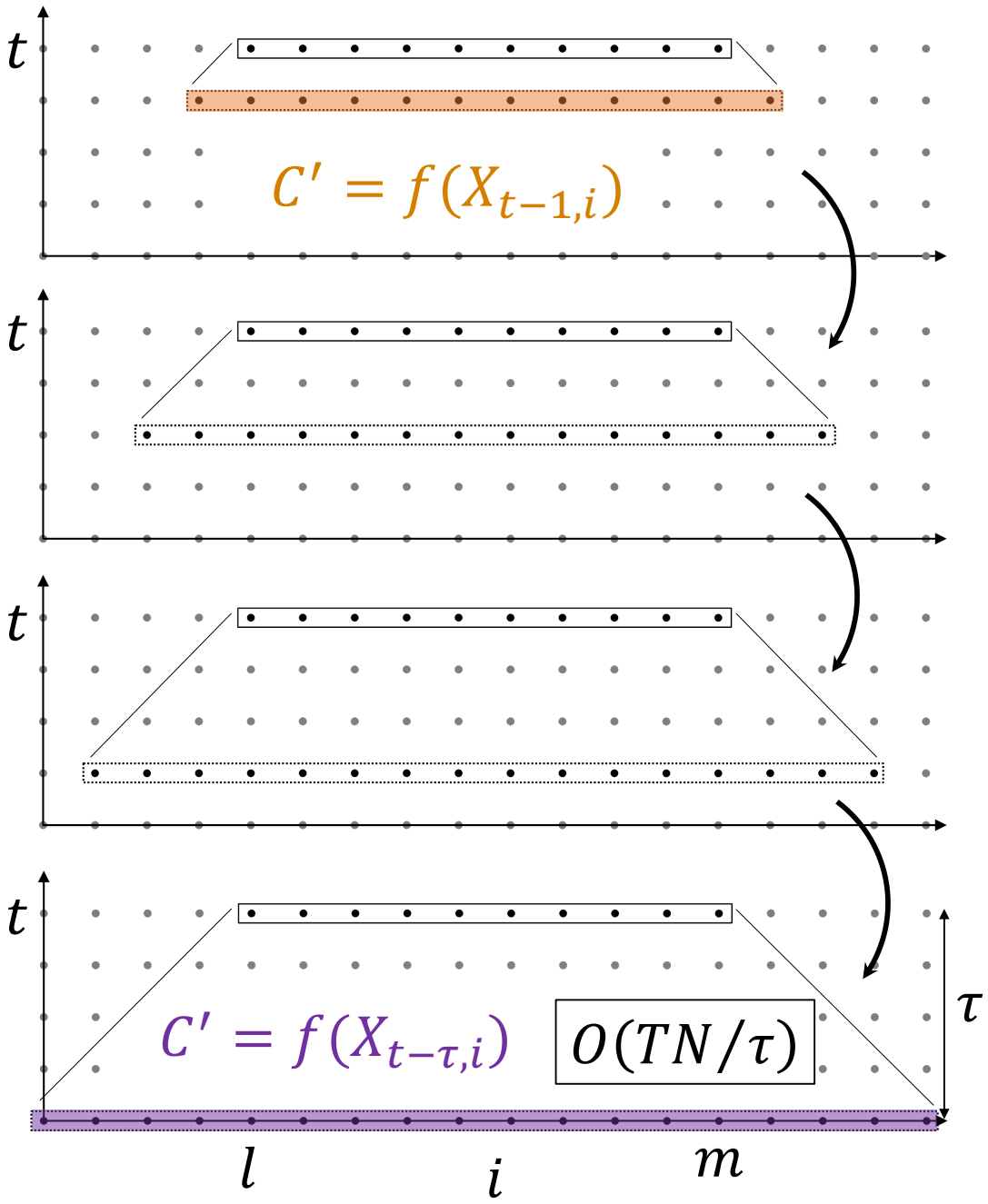


Figure 6.2: Interpolated checksum illustration after $\tau = 4$ repeated substitutions. Interpolation is more efficient than Figure 6.1.

Expressing C' as a function of $B_{t-\tau,i}$ has the same effect as what prior work [82] refers to as "interpolating" checksums solely from other checksums at previous iterations. However, we have shown the interpolation here in the opposite order (i.e., from t to $t-1$). Regardless of the order in which it is done, this process of interpolation is critically necessary to be able to carry out the checksum computations with low overhead. The intuition behind this can be understood by comparing the complexities of the non-interpolated scheme (Figure 6.1) and the interpolated scheme (Figure 6.2). The non-interpolated version is $O(TN)$ which has the same complexity up to a constant order as the stencil computation itself. The interpolated version, on the other hand, has the complexity $O(TN/\tau)$. This tunable parameter τ directly influences the complexity of checksum computation and consequently the overhead.

6.1.2 Stencils with multiple time dependencies

The interpolation process as described above is straightforward when there are only stencil dependencies from t to $t-1$, but consider the same example now with one additional dependency on $t-2$,

$$B_{t,i} = \begin{cases} A_i & \text{if } \ 0 \leq t < 2 \\ B_{t-1,i} & \text{elif } \ 2 \leq t \leq T \text{ and } i = 0 \text{ or } i = N \\ w_0 B_{t-1,i-1} + w_1 B_{t-1,i} + w_2 B_{t-1,i+1} & \\ \qquad + w_3 B_{t-2,i} & \text{else} \end{cases} \tag{6.7}$$

adapted from finite-difference time-domain (FDTD) acoustic wave simulation code [81].

In this example, the process of repeated substitution results in an alternate expression of the form,

$$C' = f(B_{t-\tau,i}) + g(B_{t-\tau-1,i}) \tag{6.8}$$

for two different functions f and g. This is because each substitution introduces terms across

the previous *two* time steps. The difficulty of identifying the correct combinations of weights here is magnified. Again, this is not something that we would want to do by hand. We show how to handle this example in Section 6.3.1 but the reader is encouraged to try to work out the equivalent version of Equation 6.4 and perform τ repeated substitutions obtaining the expressions for f and g to fully appreciate this for themselves.

6.1.3 Stencils with variable weights

Not all stencil programs are directly amenable to ABFT because it may not always be possible to do the interpolation step described in Section 6.1.1 as illustrated by the following example in Equation 6.9. The reason it was possible to rewrite Equation 6.2 as shown in Equation 6.4 was that the weight expressions were constant at each point in space allowing them to be factored out of the middle summation term.

Consider this slightly modified piece of stencil code in Equation 6.9, which is identical to Equation 6.1 except now there may be unique weight values at each point in space.

$$B_{t,i} = \begin{cases} A_i & \text{if } \ t = 0 \\ B_{t-1,i} & \text{elif } \ 0 < t \leq T \text{ and } i = 0 \text{ or } i = N \\ w_{i-1}B_{t-1,i-1} + w_iB_{t-1,i} + w_{i+1}B_{t-1,i+1} & \text{else} \end{cases} \tag{6.9}$$

It is necessary to further inspect the definition of the value of the weights, w_i before proceeding. If there exist regions with constant weights, as is common in *isotropic* stencils [51] with absorbing boundaries for example, then it is possible to carry out ABFT with low overhead. For example, isotropic stencils with absorbing boundaries typically have weights with varying values near the boundaries (i.e., close to $i = 0$ and $i = N$ in our example here)

but constant "in the middle". The definition of w_i could be of the following form,

$$w_i = \begin{cases} f(i) & 0 \leq i < M \\ 0.333 & M \leq i \leq N - M \\ f(i) & N - M < i \leq N \end{cases} \tag{6.10}$$

where f is some arbitrary non-constant function. In this case, it is possible to do ABFT, just not on the entire domain. The solution is to recognize that the third branch of Equation 6.9 can be split into three branches, one for each branch in Equation 6.10, of which the middle branch then has the desired constant-weights form. This is straightforward for a clever human to see, but much more difficult for a classical compiler that reasons at the statement level. All of these challenges are magnified in higher dimensions and beyond the most simple symmetrical stencils implementations, this is not something that we would want to do by hand. This kind of analysis is aptly suited to polyhedral compilation which enables reasoning at the statement *instance* level.

6.2 Polyhedral program representation

As discussed in Section 3.1, the polyhedral model is a mathematical formalism for reasoning about a precisely defined class of computations. In the context of ABFT, it enables us to reason precisely at the program statement instance level to construct the compact sets characterizing the hyper-trapezoidal regions illustrated previously in Figure 6.2. We use the Alpha language [14, 33, 70] to carry out most of our analysis.

In Figure 6.3, we provide an Alpha implementation of the 1D FDTD stencil from Section 6.1.2. This will be our working example for the remainder of the paper. Note the layout, with the following main sections of code: inputs, outputs, locals, and equations.

```
 0: affine fdtd [T,N,M]->{: 1<M<N and 2<T}
 1:   inputs
 2:     A : {[i] : 0<=i<=N}
 3:   outputs
 4:     A_out : {[i] : 0<=i<=N}
 5:   locals
 6:     B : {[t,i] : 0<=t<=T and 0<=i<=N}
 7:     w0,w1 : {[i] : 0<=i<=N}
 8:   let
 9:     w0[i] = case {
10:       {: 0<=i<M or N-M<i<=N } : f(i);
11:       {: M<=i<=N-M } : 0.284;
12:     };
13:     w1[i] = case {
14:       {: 0<=i<M or N-M<i<=N } : f(i);
15:       {: M<=i<=N-M } : 0.148;
16:     };
17:
18:     B[t,i] = case {
19:       {: 0<=t<2} : A[i];
20:       {: t>2 and (i=0 or i=N)} : B[t-1,i];
21:       {: t>2 and 0<i<N} : w0[i-1]*B[t-1,i-1]
                            + w0[i]*B[t-1,i]
                            + w0[i+1]*B[t-1,i+1]
                            + w1[i]*B[t-2,i];
22:     };
23:     A_out[i] = B[T,i];
24: .
```

Figure 6.3: 1D FDTD Alpha program with multiple time dependencies (from Section 6.1.2) and regions with variable weights (Section 6.1.3).

6.2.1 Checksums as Alpha reductions

We use reductions to represent the checksum values over the program variables in the following sections. Equation 6.2 can be expressed in Alpha as,

```
C[t,l,m] = reduce(+, (t,l,m,i->t,l,m),
                                {: l<=i<=m } : B[t,i])
```

for some new variable C representing the *family* of all possible checksum *instances* at a particular time step t over a particular range of i for $l \leq i \leq m$. We need the family to fully cover the program as discussed in Section 6.3.2. The reduction body here is a restrict expression with a 4-dimensional expression domain over the indices t, l, m, and i. At each

point in this domain, the program variable B is read at $B[t, i]$ and accumulated into the checksum instance at $C[t, l, m]$.

In this context, the objective is to transform this reduce expression into the equivalent efficient expression over the base of the corresponding trapezoidal domain in Figure 6.2. Alpha supports algebraic substitution and simplification of individual expressions, which makes this possible.

6.3 Automatic Checksum Derivation

To facilitate the presentation we make the following assumptions about the input stencil program,

- the outer dimension on stencil variables is interpreted as the index over time
- the equations characterizing the stencil body update points at time step t from points at strictly earlier time steps (i.e., $t - k$ for some constant $k > 0$)
- the dimensionality of all stencil variables is the same

This introduces no loss in generality because it is always possible to achieve this by reindexing the program variables. For example, consider the following Gauss-Seidel stencil equation,

$$X_{t,i} = aX_{t,i-1} + bX_{t-1,i} + cX_{t-1,i+1} \tag{6.11}$$

with the dependency across the same time step from $[t, i]$ to $[t, i - 1]$. In such cases, we can reindex X such that all dependencies point strictly backward in time with the mapping $\{[t, i] \rightarrow [2t + i, 2i]\}$ as illustrated in Figure 6.4. The interested reader can notice that this is the task of finding a set of valid scheduling hyperplanes [25].

With these assumptions in mind, principally our approach involves the following three steps to automate ABFT,

1. constructing checksums via algebraic substitution

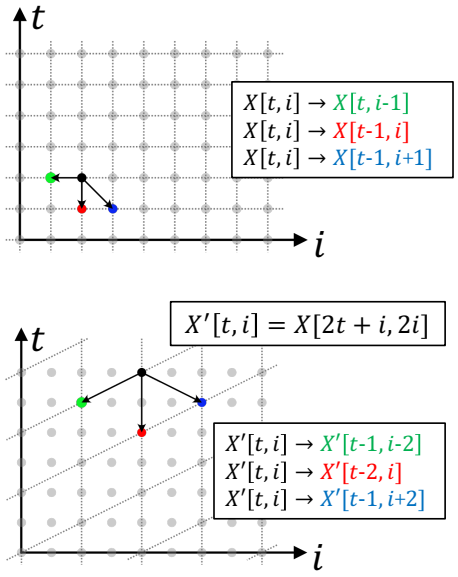


Figure 6.4: Normalization of dependencies in Equation 6.11 (top) such that all dependencies subsequently point strictly backward along time t (bottom).

2. replicating checksums over the program domain

3. scheduling and code generation

which are discussed in detail in the following subsections.

6.3.1 Step 1 - Construct invariant checksum with computer algebra

Given an input stencil program like the one provided in Figure 6.3, the goal of this step is to express the stencil in such a way that we can systematically build, and subsequently insert, two algebraically equivalent expressions computing the same checksum.

Merge variables

Some preprocessing may be required. We will call two stencil variables, X and Y, *coupled* if they belong to the same strongly connected component in the program dependency graph. For example, if the equation for X has an expression involving Y and the equation for Y in turn has some expression involving X, then we will say that X and Y are *coupled*. There

could be more than two variables as well. For example, imagine that X depends on Z which depends on Y. In this case we will say that X is coupled with both Y and Z and we will refer to the set of coupled variables as $\{X, Y, Z\}$. When there exist coupled variables like this, we merge them into a single variable as follows. Let $X_i[s]$ be a dependency expression on the i'th such coupled variable. First, create a new higher dimensional variable X' with an additional dimension. Then embed the equations for each variable X_i in X' by associating the i'th index value in this additional dimension with X_i. This is purely a syntactic rewrite and is always legal.

Many user-written stencil codes simulating electromagnetism, for example, contain multiple stencil variables. Consider the following example, adapted from other FDTD codes in the MathWorks library [93], with separate variables for the electric (E) and magnetic (H) fields,

$$E_{t,i} = \begin{cases} E_{t-1,i} & t > 0 \text{ and } i = 0 \\ E_{t-1,i} - a_i(H_{t-1,i} - H_{t-1,i-1}) & t > 0 \text{ and } 0 < i \leq N \end{cases}$$

$$H_{t,i} = \begin{cases} H_{t-1,i} - b_i(E_{t-1,i+1} - E_{t-1,i}) & t > 0 \text{ and } 0 \leq i < N \\ H_{t-1,i} & t > 0 \text{ and } i = N \end{cases}$$

which can be merged into a single variable (M) as,

$$M_{t,i,z} = \begin{cases} M_{t-1,i,z} & \text{...and } z = 0 \\ M_{t-1,i,z} - a_i(M_{t-1,i,z+1} - M_{t-1,i-1,z+1}) & \text{...and } z = 0 \\ M_{t-1,i,z} - b_i(M_{t-1,i+1,z-1} - M_{t-1,i,z-1}) & \text{...and } z = 1 \\ M_{t-1,i,z} & \text{...and } z = 1 \end{cases}$$

with the same piecewise constraints associated with the new index z. Here z is the index associated with the new dimension, and since there are two coupled variables, z only takes

on two values. The original variable E is associated with $z = 0$ and H with $z = 1$.

Normalize Sum of Products Expressions

Expressions for each merged stencil variable, X, and its weights, w_i are transformed into expressions of the form,

$$w_1[f_1(z)] * X[f_1(z)] + w_2[f_2(z)] * X[f_2(z)] + ... \tag{6.12}$$

denoting a normal **S**ums **o**f **P**roducts (SoP) expression, where the left-hand side of each product is an expression for the weights and the right-hand side is the stencil variable. For example, the expression,

```
X[t-1,i] - a[i]*(X[t-1,i+1] - X[t-1,i])
```

is rewritten as the following normal SoP expression,

```
1*X[t-1,i] + (-1*a[i])*X[t-1,i+1] + a[i]*X[t-1,i]
```

This is purely a preprocessing step for the next step to identify the regions with constant weights.

Identify convolution domains

Given two Alpha variables X and W, let us define the convolution of W with X by the operator $\bigotimes$ at the point $s \in \mathcal{D}_X$ as[2],

$$W \bigotimes X_s = \sum_{s' \in \mathcal{D}_W} W_{s'} X_{s+s'} \tag{6.13}$$

Given an SoP expression involving X as defined in Equation 6.12, we construct a definition for W. We want to identify the subdomain $\mathcal{P}$ that reads the same value from each weight expression appearing in the SoP. Then we define the value of $W[s]$ for each $s \in \mathcal{P}$ as

[2]The symbol "$*$" is commonly used elsewhere in the literature to denote convolutions, but to avoid conflating it with Alpha's multiplication, we use $\bigotimes$ for convolutions.

the corresponding constant weight subexpression. Then we rewrite the portion of the SoP in $\mathcal{P}$ as the convolution defined in Equation 6.13.

As an example, consider the following equation,

$$X[i] = w * X[i-1] \tag{6.14}$$

for the 1D variable X with the domain $\mathcal{D}_X$. The binary expression $w*X[i]$ on the right-hand side is defined over $\mathcal{D}_X$, and the subexpression w is read at each point in this domain. The subexpression w here should be understood as the dependence expression that reads the scalar variable w at each point in the expression domain by the affine function $f : [i] \to []$. We say that *there is reuse of w in context* of its use because the null space of f has a non-empty intersection with the expression domain[3]. This notion of reuse in context is precisely the kind of resuse discussed previously relating to simplification [49]. We leverage this to identify the reuse space common to all products in the SoP expression.

Looking again at the body of the restrict expression on line 21 of Figure 6.3, this reuse analysis can be run on each of the four expressions in a bottom-up fashion. The context domain $\mathcal{D}_{\text{context}}$ of this expression is $\{[t,i] : 0 \leq t \leq T \text{ and } 0 < i < N\}$, taken from its parent restrict expression. The first weights expression $w0[i-1]$ is only constant in the subdomain $\{[t,i] : 0 \leq t \leq T \text{ and } M \leq i-1 \leq N-M\}$. All four weights expressions are simultaneously constant in the subdomain $\mathcal{P} = \{[t,i] : 0 \leq t \leq T \text{ and } M < i < N-M\}$.

From here, we split the case branch containing this SoP into two cases. In this example, we are left with the following two expressions,

```
{: t>2 and (0<i<=M or N-M<=i<N)} :
        w0[i-1]*B[t-1,i-1] + w0[i]*B[t-1,i]
    + w0[i+1]*B[t-1,i+1] + w1[i]*B[t-2,i];
{: t>2 and M<i<N-M} :  // the convolution
        0.284*B[t-1,i-1] + 0.284*B[t-1,i]
```

[3]Note that we use the `isl`[62] notation to indicate that f maps onto a 0-dimensional space denoted by the empty tuple [].

```
    + 0.284*B[t-1,i+1] + 0.148*B[t-2,i];
```

The second expression here with constant weights can be seen as the convolution $W \bigotimes B_s$ where $W[-1,-1]$, $W[-1,0]$, and $W[-1,1]$ are 0.284 and $W[-2,0]$ is 0.148.

Stencil as a composed convolution

At this point, we have identified subdomains of the stencil variables that are computed as convolutions of the form,

$$X_s = W \bigotimes X_s \tag{6.15}$$

From this definition, we can substitute X_s *one time* on the righthand side with Equation 6.15 to obtain,

$$X_s = W \bigotimes \Big(W \bigotimes X_s \Big) \tag{6.16}$$

or more generally τ *times* leaving,

$$X_s = W \bigotimes \Big(W \bigotimes \quad ... \quad \bigotimes \Big(W \bigotimes X_s \Big) \Big) \tag{6.17}$$

Since convolutions are, by definition, commutative and associative, this can be equivalently expressed as,

$$X_s = W^\tau \bigotimes X_s \tag{6.18}$$

where the convolution kernel first applied to itself τ times to produce a larger kernel, W^τ which is then applied to the variable X at each point s. Note that Equations 6.15 and 6.18 compute the same value. However, the subset of points read from X differs between the two, which can be seen by referring back to Equation 6.13.

Construction of the invariant checksums

For each stencil variable X defined over the domain $\mathcal{P}$ and its corresponding composed convolution expression defined in Equation 6.18, we construct one checksum expression pair C and C'. Let C be defined as the following reduction over points in $\mathcal{Q}$, a rectangular

subdomain of $\mathcal{P}$,

$$C = \sum_{s \in \mathcal{Q}} X_s \tag{6.19}$$

Let C' denote an alternate checksum, which computes the same value, using Equation 6.18,

$$C' = \sum_{s \in \mathcal{Q}} \left(W^{\tau} \bigotimes X_s \right) \tag{6.20}$$

The points from X contributing to C are separated by τ timesteps from the points contributing to C'. Geometrically, these are the "top" and "bottom" faces of hyper-trapezoidal regions, as illustrated in Figure 6.2 for the 2D case. Silent errors occurring at any point inside this hyper-trapezoid result in different computed values for C and C'.

6.3.2 Step 2 - Program Error Protection Coverage

Now we have a single checksum expression pair for each convolution appearing in the normalized program. However, each pair only enables the detection of errors within their corresponding hyper-trapezoidal subdomain. In order to use this over the rest of the convolution, we fix the trapezoid to a constant (non-parametric) size from the family of all possible checksums and then stride it over the convolution domain (very similar to tiling for all intents and purposes) as shown in Figure 6.5.

Still, it is not possible to cover every point. Points in the red region in Figure 6.5 need to be handled with some other error detection scheme because they are not contained by any trapezoid. We can use duplication here, which is not a concern from an overhead perspective since this only occurs on the boundaries with an asymptotically smaller complexity. Let $\mathcal{D}_B$ denote the domain of the stencil variable B defined in line 6 of Figure 6.3. Let $\mathcal{D}_{ABFT}$ denote the domain of the union of all trapezoids. Let $\mathcal{D}_{dup}$ denote the duplication domain and be defined as, $\mathcal{D}_{dup} = \mathcal{D}_B \setminus \mathcal{D}_{ABFT}$ where "\" denotes set difference. Then we add another program variable, called B_dup, using the same equation as B but restricted to $\mathcal{D}_{dup}$.

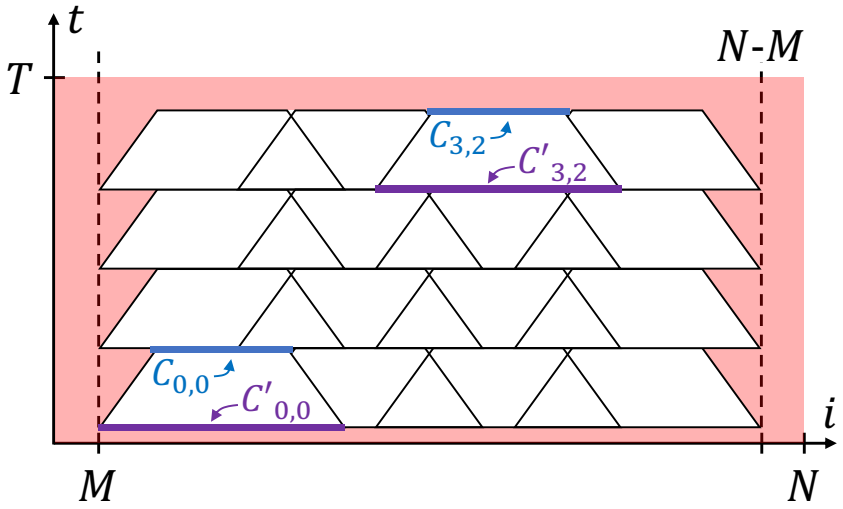


Figure 6.5: Striding checksum expression pairs over the corresponding convolution domain, shown here for the example in Figure 6.3. The subscripts $[ct, ci]$ here denote the ct'th row (from bottom to top) and ci'th column. The shaded red region illustrates the domain where duplication is needed.

6.3.3 Step 3 - Scheduling and code generation

At this point, the original stencil computation and the checksum expression pairs are completely decoupled. The stencil variables and checksum expression pairs need to be computed in a lock-step fashion because practical stencil implementations only use as much memory as required for two time steps' worth of computation[4], then with "modulo 2" accessing used on the time dimension. Points in the checksums need to be accumulated into C and C' before they are overwritten at the next time step iteration. We can construct the schedule where at each time step, we first update the stencil variables, followed by any corresponding checksum variables if they exist, followed by duplication regions. Note that many time steps have no updates to any checksum variables since updates only occur at the bottom and top of the trapezoidal tiles. Alpha can be used to generate C code.

[4]If there are time dependencies back to $t - k$, then k time steps' worth of memory is required of course.

6.4 Evaluation

In this section, we provide a qualitative and quantitative discussion of the effectiveness of our approach. Our primary contribution is automation so we emphasize the applicability of our analysis on non-trivial user-written stencil codes.

6.4.1 Qualitatitve Evaluation

As alluded to in Section 6, there are many factors that influence the particular form of a given stencil program, which are often more complicated than the trivial Jacobi stencils. We discuss two such stencil applications taken from GitHub and the MathWorks library and show how our approach can be applied. The first example simulates acoustic wave propagation in 2D [81]. The second example simulates electromagnetic field propagation in 3D using FDTD methods [93]. Both examples deal with non-trivial boundary conditions and are implemented in MatLab. It is worth noting that the semantics of MatLab resembles that of Alpha, in the sense that one does not need to specify the surrounding point loops and can, instead, write element-wise operations to entire data arrays with a single statement.

2D wave propagation code with absorbing boundaries

The first stencil application we consider implements a 2D wave propagation simulation [81]. Due to space constraints, we only highlight, with pseudo-code, the key pieces critical to our analysis. That said, this code has three parts. The corresponding Alpha implementation has equations like those shown in Figure 6.6. The references to Nx and Nz can be understood as size parameters of the stencil domain. Similarly, Lx and Lz are the size parameters of the subdomain where the absorbing boundary conditions hold. MatLab's element-wise operations "`.*`" between the variables, `p` and `weights`, is naturally expressed in Alpha as "`p[t-1,z,x] * weights[z,x]`" where the points in `p` at timestep $t-1$ represent the entire `p2` array. The core challenge here is due to the fact that the expressions appearing in the stencil loop for the weights variable do not read the same value at every point, which means

that the convolution can not be directly inferred. In this example, our approach will use the analysis in Section 6.3.1 to identify the interior convolution domain shown in the bottom of Figure 6.6. Once this convolution domain is identified, we have all the information we need in order to apply the existing ABFT methodologies for stencils. In this sense, we can view this convolution as a variant of the simple 2D star stencil as illustrated in Figure 6.8.

```
// Computed variable
outputs
  p : {[T,Nz,Nx]}
let
// Absorbing boundaries
weights[z,x] = case {
  {: z<Lz or Nz–Lz<z } : f(z);
  {: x<Lx or Nx–Lx<x } : f(x);
  {: Lx<=x<=Nx–Lx and Lz<=z<=Nz–Lz } : 1;
}
// Main stencil loop
// p1 = p2 .* weights
p[t,z,x] = p[t-1,z,x] * weights[z,x]
```

```
// Rewritten main stencil loop
p[t,z,x] = case {
  {: z<Lz or Nz–Lz<z or x<Lx or Nx–Lx<x} : p[t-1,z,x] *
                                           weights[z,x];
  {: Lx<=x<=Nx–Lx and Lz<=z<=Nz–Lz} : 1 * p[t-1,z,x];
}
```

Figure 6.6: Alpha components of the 2D wave propagation MatLab code. Given the code snippet on top, our analysis produces the code snippet at the bottom where the convolution can be identified.

3D FDTD electromagnetic wave simulation

The second example we consider is a significantly more complicated application simulating electromagnetic wave propagation in 3D [93]. Like the previous wave propagation example, this also has complex boundary conditions where the various weight variables are not constant in the regions close to the boundaries. These can be handled in the same way. In this example, there is additional complexity. Note that there are six stencil variables in total, the electric and magnetic field components for each of the physical spatial dimensions: E_x, E_y, E_z, H_x, H_y, and H_z. Note the intertwined dependencies across the variables (e.g., H_x

reads values from E_z which in turn reads from H_y, and so on). Due to space constraints, we

```
// Computed variable
outputs
  Ez : {[T,Nx,Ny,Nz]}
let
// Main stencil loops
Hx[t,i,j,k] = case {
  D1 : Hx[t-1,i,j,k];
  D2 : ((Hx[t-1,i,j,k] + (Chxey[] * (Ey[t-1,i,j,k+1] -
       Ey[t-1,i,j,k]))) - (Chxez[] * (Ez[t-1,i,j+1,k] -
       Ez[t-1,i,j,k])));
};
// Hy, Hz, Ex, Ey equations not shown
...
Ez[t,i,j,k] = case {
  D11 : Ez[t-1,i,j,k];
  d12: ((Ez[t-1,i,j,k] + (Cezhy[i,j,k] * (Hy -
       Hy[t,i-1,j,k]))) - (Cezhx[i,j,k] * (Hx -
       Hx[t,i,j-1,k])));
};
```

```
// Merged single stencil variable
M[t,i,j,k,z] = W[i-1,j,k,0] * M[t-1,i-1,j,k,0] +
               W[i+1,j,k,1] * M[t-1,i+1,j,k,1] +
               W[i,j-1,k,2] * M[t-1,i,j-1,k,2] +
               W[i,j+1,k,3] * M[t-1,i,j+1,k,3] +
                ...;
W[i,j,k] = case {
  {: z=0 and D0'} : 1 + Cexhz[i,j,k] - Chzex[] + ...;
  {: z=1 and D1'} : 1 + Ceyhx[i,j,k] - Chyex[] + ...;
  ...;
}
```

Figure 6.7: Alpha components of the 3D FDTD code. Given the code snippet on top, we obtain code in the form shown at the bottom.

do not list the complete normalized version of the code, but after merging all six variables as described in Section 6.3.1, we are left with a program in the form of the bottom code snippet in Figure 6.7. Note that this is now in a form consumable by the analysis in Section 6.3.1 to identify the interior convolution domain. In the end, we are left with a subdomain over which we have inferred a convolution. Again, the shape of this convolution is a higher dimensional version of the star stencils illustrated in Figure 6.8.

6.4.2 Quantitative Evaluation

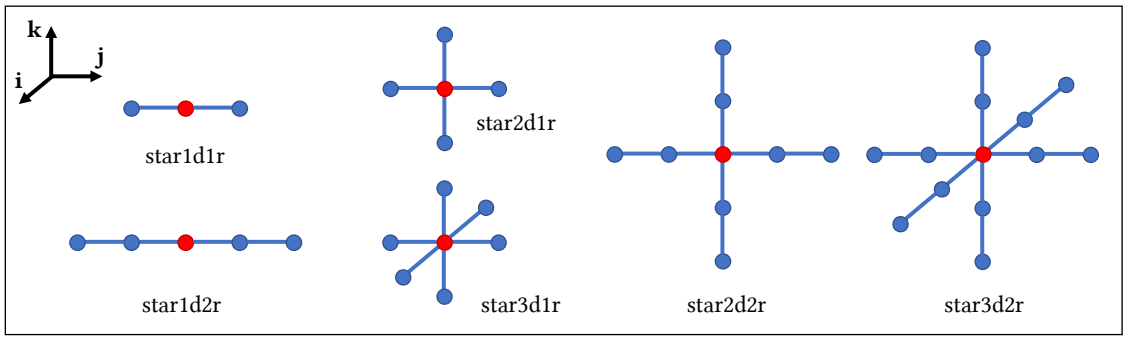


Figure 6.8: The core stencil pattern present in the resulting convolutions. The notation "X"d"Y"r should be read as "X" dimensional with a radius of "Y".

In this section, we report data on the overhead of the generated code produced by our analysis. Overhead refers to the execution time of the checksum-augmented program relative to the corresponding input program. We emphasize that the core contribution of this work is focused on automation and the machinery required to identify and generate code for the subdomains carrying out convolutions. Once the convolution subdomains have been identified, we can associate them with a corresponding version of the star stencil pattern illustrated in Figure 6.8.

All experiments were conducted on a machine running Linux with a 12th Gen Intel Core i7-12700K processor and 64Gb of memory. We report results for six different stencil codes. For several combinations of length L and height H listed in Table 6.1, we generated two versions of C code implementing the same stencil computation. One that only carries out the stencil computation, and one that carries out the stencil computation along with its corresponding checksum computations. In Figure 6.9, we report the relative performance of the checksum augmented code versus the baseline. We use L to denote the size of the base side length, and H to denote the height. In all cases, we only report data for hyper-trapezoidal shapes with square aspect ratios along the spatial dimensions, though this is not a limitation of our approach. For values of H near 16, we incur overhead near 10%. This corresponds to the results obtained from a manual implementation of a similar ABFT

Version	H	L	T	N
star1d1r	[2,4,8,12,16]	[50..200..50]	1000	4×10^9
star1d2r	[2,4,8,12,16]	[50..200..50]	1000	4×10^9
star2d1r	[1,2,4,8,12,16]	[40..200..40]	500	5000
star2d2r	[1,2,4,8,12,16]	[40..200..40]	500	5000
star3d1r	[1,2,4,8,12,16]	[20..100..20]	500	500
star3d2r	[1,2,4,8,12,16]	[20..100..20]	500	500

Table 6.1: Hyper-trapezoidal shape configuration explored for overhead experiments. The notation "[a..b..c]" denotes the inclusive range of values between "a" and "b" strided by "c".

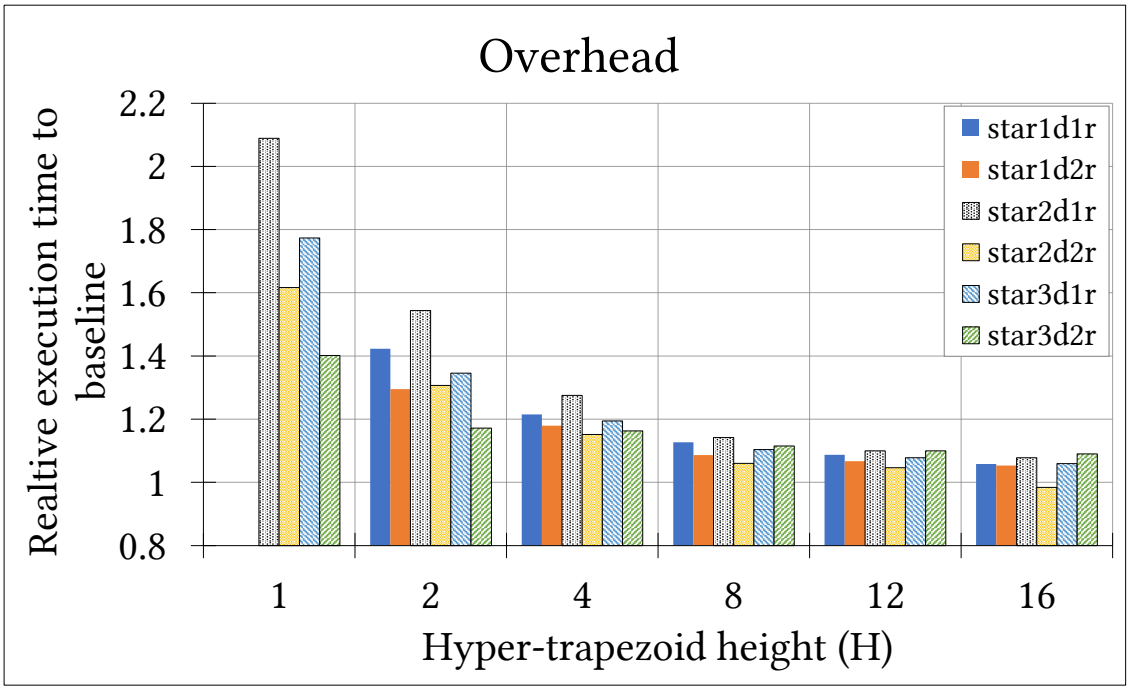


Figure 6.9: Shows the overhead as a function of the height H. As we expect, and was shown by prior work, larger heights lead to lower overhead.

approach by Cavelan and Ciorba [82]. Our conclusion is that the automated code generation is competitive.

In addition to performance issues, an important concern for ABFT schemes in general is error sensitivity. Although it is an important issue, it is orthogonal to the automation process discussed in this work (we invite the interested reader to refer to the work of Cavelan and Ciorba [82] for a more in-depth discussion of this aspect). Nevertheless, for the sake of completeness, we have evaluated the sensitivity of the error detection capability on a simple 1D stencil and report our findings in Figure 6.10. We simulated errors with random single

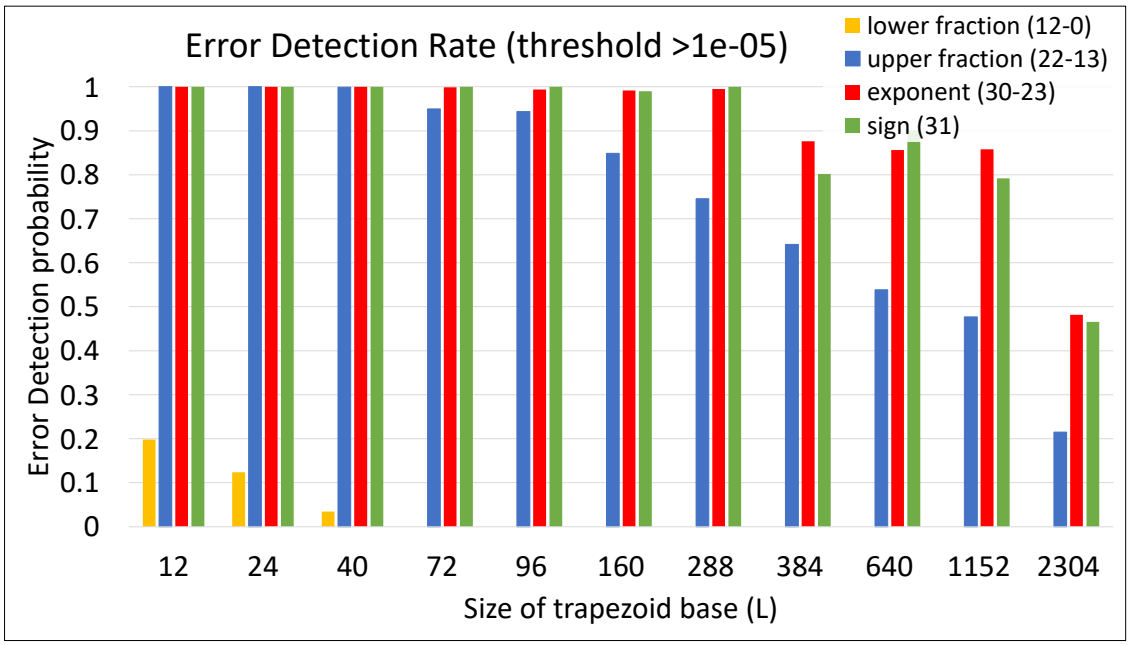


Figure 6.10: Error injection experiment for the star1d1r example as a function of the base trapezoidal volume (i.e., number of points contributing to C' in Equation 6.20.

bit flips on the array holding the primary stencil data in memory during execution of the program. To collect the data for Figure 10, we used a single precision (32-bit) data grid size of $N = 12000$ updated over $T = 100$ time steps. For each of the 32 bit flip positions, we ran 100 trials. If there was at least one checksum expression pair with a difference above the threshold of 10^{-5} then the run was counted as an error detection success. The fraction of total successes is reported for each checksum size (L) and bit flip position grouped by sign, exponent, upper, and lower fraction. Modeling faults as bit flips that propagate through the computation is the standard error-injection simulation approach used by related work on ABFT. As expected and also reported by Cavelan and Ciorba [82], we observe that as the size of the checksum region increases, it becomes harder to differentiate the presence of an error from floating-point round-off errors intrinsically present in the checksum computations. However, exploring this trade-off in depth is out of the scope of this work.

6.5 Related Work

With any fault tolerance scheme, there is typically a trade-off between error coverage and overhead. Duplication and related schemes like Triple Modular Redundancy [1], for example, have the highest error coverage but are often too costly. Cavelan and Ciorba showed how to use ABFT on stencil computations, on effectively the star3d1r stencil shown in Figure 6.8, via interpolating checksum expressions over multiple time steps [82]. Their approach requires very careful and manual human analysis, however, and can not handle stencils with multiple time dependencies as with the example in Section 6.1.2. We report overhead and error detection results in agreement with their findings as a sanity check. Their work is a rediscovery of an even older work on ABFT in the context of stencils by Roy-Chowdhury et. al. [36]. Our work can be viewed as a generalization of these two approaches with a wider range of applicability.

There is other prior work that employs fault tolerance analysis at compile time. Tavarageri et. al. proposed a compiler-assisted detection of memory errors [73]. Their approach works by ensuring that any data written to memory locations remain unchanged during any of its subsequent uses. This is achieved by augmenting the input program with checksums at compile time to keep track of the values *written into* and subsequently *read from* memory. Their approach has low overhead but has a lower error coverage than ABFT approaches. Errors that occur outside of memory, in the floating point arithmetic units, for example, go undetected. Our analysis here is based on numerical properties of the computed data which means we can detect errors *anywhere* in the pipeline as long as they are not swallowed by floating-point round-off noise.

6.6 Open Questions and Future Work

There is much more than can be explored here with respect to error detection accuracy. Our primary contribution is the automation behind the analysis and construction of ABFT

checksum computations. Our approach relies on identifying regions in the stencil domain computed using convolutions. However, these weight expressions in the stencil's convolution kernel may not be constant at each point in space or time. Stencils of this form correspond to the class of *anisotropic* partial differential equations [51]. If the weights truly are unique at each point in space, then efficient interpolation of checksums across time steps is not possible. This is because the weights can not be factored out of the expression in Equation 6.4. To the best of our knowledge, there has been no prior work on how to carry out ABFT efficiently on such anisotropic stencil applications and this remains an open and interesting problem.

6.7 Conclusion

We have studied the use of ABFT for stencil computations and proposed a new technique to automatically augment the input program with checksums to detect the occurrence of silent transient errors. We show that low overhead code can be easily generated thanks to polyhedral analyses and our preliminary results illustrate that there is an interesting trade-off worth exploring further.

Chapter 7

Extending ABFT Schemes in Stencil Computations

In the previous chapter, we showed how the application of ABFT can be automated for stencil computations. Our primary contribution was around automation, specifically how to carry out, at compile time, the program analysis required to construct checksums as a function of the input program dependencies. As such, we provided some preliminary results on the performance of the ABFT-augmented input stencil programs. In this chapter, we consider several different ways to carry out ABFT in stencils, providing a more thorough analysis of the performance as a function of the size and shape of the ABFT checksum regions.

In the context of ABFT, *performance* has two independent components. The first of these is the *overhead*—the ratio of the execution time of the checksum-augmented program to the input program. The second is the *detection rate*—the percentage of the time we can correctly measure checksum differences above some threshold when simulated errors (bit flips) are injected. Both are affected by the shape and size of the checksum regions (i.e., the hyper-trapezoidal domains illustrated previously previously). It is challenging to construct the checksum regions in such a way that both minimizes the overhead and maximizes the detection rate. From Figures 6.9 and 6.10 from Chapter 6, we can see that as the size of the checksum regions increases (i.e., increasing the trapezoidal height, H), the overhead decreases. However, once the checksum region grows larger than 2000 points, the detection rate drops significantly. For 1D stencils, our previous approach works well. We can construct ABFT-hardened implementations where the overhead is sufficiently small while also maintaining good detection rates. However, our previous approach has difficulty satisfying both objectives when the stencil data involves two or more spatial dimensions. For all possible checksum size and shape configurations, the ones that incur overheads of less than 2 have

horrible detection rates. Conversely, the configurations with reasonable detection rates have high overheads.

This chapter discusses three different ABFT checksum schemes for stencils and compares and evaluates their effectiveness. In doing so, we show how to augment previously proposed automation to construct ABFT-hardened programs with *both* low overhead and reasonable detection rates for 1D, 2D, and 3D stencil variants. As with prior work, we implemented all three approaches in the AlphaZ system.

7.1 Background and Working Example

Let us consider the same Jacobi 1D stencil discussed previously, which updates an $(N+1)$-element array over a series of T time steps, shown again below.

$$B_{t,i} = \begin{cases} A_i & \text{if } \ t = 0 \\ B_{t-1,i} & \text{elif } \ 0 < t \leq T \text{ and } i = 0 \text{ or } i = N \\ w_0 B_{t-1,i-1} + w_1 B_{t-1,i} + w_2 B_{t-1,i+1} & \text{else} \end{cases} \tag{7.1}$$

In this example, the weights are constants, and the time dependencies are only from t to $t-1$ (i.e., this is an isotropic first-order stencil). Recall that the way ABFT works for stencils is to define two algebraically equivalent expressions that both compute the same value using different program slices. To do this, we introduce a new variable, $C_{t,l,m}$, to denote the checksum at time step t defined as the sum over the window of values of $B_{t,i}$ for $l \leq i \leq m$,

$$C_{t,l,m} = \sum_{i=l}^{m} B_{t,i} \qquad 1 \leq l \leq m \leq N-1 \tag{7.2}$$

Then we introduce another new variable $C'_{t,l,m}$, which is obtained by repeated substitution of Equation 7.1 in Equation 7.2. The challenge is that, in general, repeated substitution can lead to an explosion in the number of terms appearing on the right-hand sides of our equations. However, as we discussed in Section 6.3.1, stencils may be viewed as instances

of convolutions leading to a very regular pattern. In this example, we can identify the convolution kernel W as,

$$W_i = \begin{cases} w_0 & i = -1 \\ w_1 & i = 0 \\ w_2 & i = 1 \end{cases} \tag{7.3}$$

Using $\bigotimes$ as the convolution operation defined given previously in Equation 6.13, we may write down a simple closed form expression equivalent to τ repeated substitutions,

$$C'_{t,l,m} = \sum_{i=l}^{m} \left(W^{\tau} \bigotimes B_i \right) \tag{7.4}$$

This scheme is illustrated again in Figure 7.1. When the values of $C_{t,l,m}$ and $C'_{t,l,m}$ are sufficiently large, we can conclude that an error occurred at some iteration point inside this trapezoidal region. We will refer to this scheme as *complete inlining*, in the sense that the form of Equation 7.4 follows from repeatedly substituting *all* occurrences of the stencil variable B that are introduced during each successive substitution. Note that this

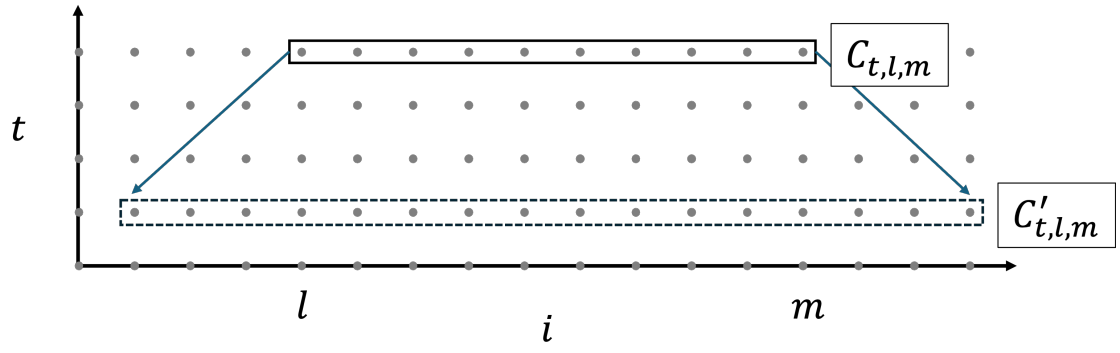


Figure 7.1: Checksum domains C and C' obtained from *complete* repeated substition.

generalizes to higher dimensions. In the analogous d-dimensional stencil, the dimensionality of the checksum domains (i.e., the domains characterizing which points in the iteration space belonging to the top and bottom of their corresponding *hyper*-trapezoids) is also d-

dimensional.

There is no reason that we must carry out repeated substitutions *completely*, nor is it necessary to do so along every dimension. In the following sections, we will discuss two alternate ways to construct $C'_{t,l,m}$.

7.2 New Checksum Scheme - Partial Inlining

Let us introduce an alternative way to construct $C'_{t,l,m}$. We begin by making a single substitution of Equation 7.1 in Equation 7.2. Recall that after some algebra, grouping terms based on the unique combinations of the weights, we can obtain the following equation,

$$\begin{aligned} C'_{t,l,m} \equiv\ & (w_0)B_{t-1,l-1} + (w_0 + w_1)B_{t-1,l} \\ & + (w_0 + w_1 + w_2)\sum_{i=l+1}^{m-1} B_{t-1,i} \\ & + (w_1 + w_2)B_{t-1,m} + (w_2)B_{t-1,m+1} \end{aligned} \tag{7.5}$$

So far, this is identical to the previously proposed method. However, the key observation that differentiates this scheme is to recognize that the middle summation term is actually $C_{t-1,l+1,m-1}$, a smaller version of the same checksum variable at the previous time step. This allows us to express C' recursively as the following,

$$\begin{aligned} C'_{t,l,m} =\ & (w_0)B_{t-1,l-1} + (w_0 + w_1)B_{t-1,l} \\ & + (w_0 + w_1 + w_2)C'_{t-1,l+1,m-1} \\ & + (w_1 + w_2)B_{t-1,m} + (w_2)B_{t-1,m+1} \end{aligned} \tag{7.6}$$

We can expand this further by inlining the definition $C'_{t-1,l+1,m-1}$ and again performing algebra to collect terms based on the patterns of weight combinations they share to obtain,

$$\begin{aligned} C'_{t,l,m} = \; & (w_0)B_{t-1,l-1} + (w_0)(w_0+w_1+w_2)B_{t-2,l} \\ & + (w_0+w_1)B_{t-1,l} + (w_0+w_1)(w_0+w_1+w_2)B_{t-2,l+1} \\ & + (w_0+w_1+w_2)^2 C'_{t-2,l+2,m-2} \\ & + (w_1+w_2)B_{t-1,m} + (w_1+w_2)(w_0+w_1+w_2)B_{t-2,m-1} \\ & + (w_2)B_{t-1,m+1} + (w_2)(w_0+w_1+w_2)B_{t-2,m} \end{aligned} \tag{7.7}$$

Like the previous definition of C', from Equation 7.5, repeated substitution leads to a very regular pattern. The terms in Equation 7.7 are grouped to illustrate the pattern that emerges from this regularity. With each additional inlining of C', one additional term gets added to each of the four weights combos, w_0, w_0+w_1, w_1+w_2, and w_2, multiplied by another factor of $w_0+w_1+w_2$. We show the algebra here to provide the intuition but note that this takes place on the Alpha IR, and we can effectively coalesce these points into single convex sets, as shown at the bottom of Figure 7.2, and implement these as reductions. The colorings shown in the figures match the corresponding colored text in the equations. After τ substitutions, we can write the following closed-form representation of each of the oblique regions in Figure 7.2 as,

$$\begin{aligned} C'_{t,l,m} = \; & (w_0) \sum_{0 \le p < \tau} (w_0+w_1+w_2)^p B_{t-1-p,l-1+p} \\ & + (w_0+w_1) \sum_{0 \le p < \tau} (w_0+w_1+w_2)^p B_{t-1-p,l+p} \\ & + (w_0+w_1+w_2)^\tau C'_{t-\tau,l+\tau,m-\tau} \\ & + (w_1+w_2) \sum_{0 \le p < \tau} (w_0+w_1+w_2)^p B_{t-1-p,m-p} \\ & + (w_2) \sum_{0 \le p < \tau} (w_0+w_1+w_2)^p B_{t-1-p,m+1-p} \end{aligned} \tag{7.8}$$

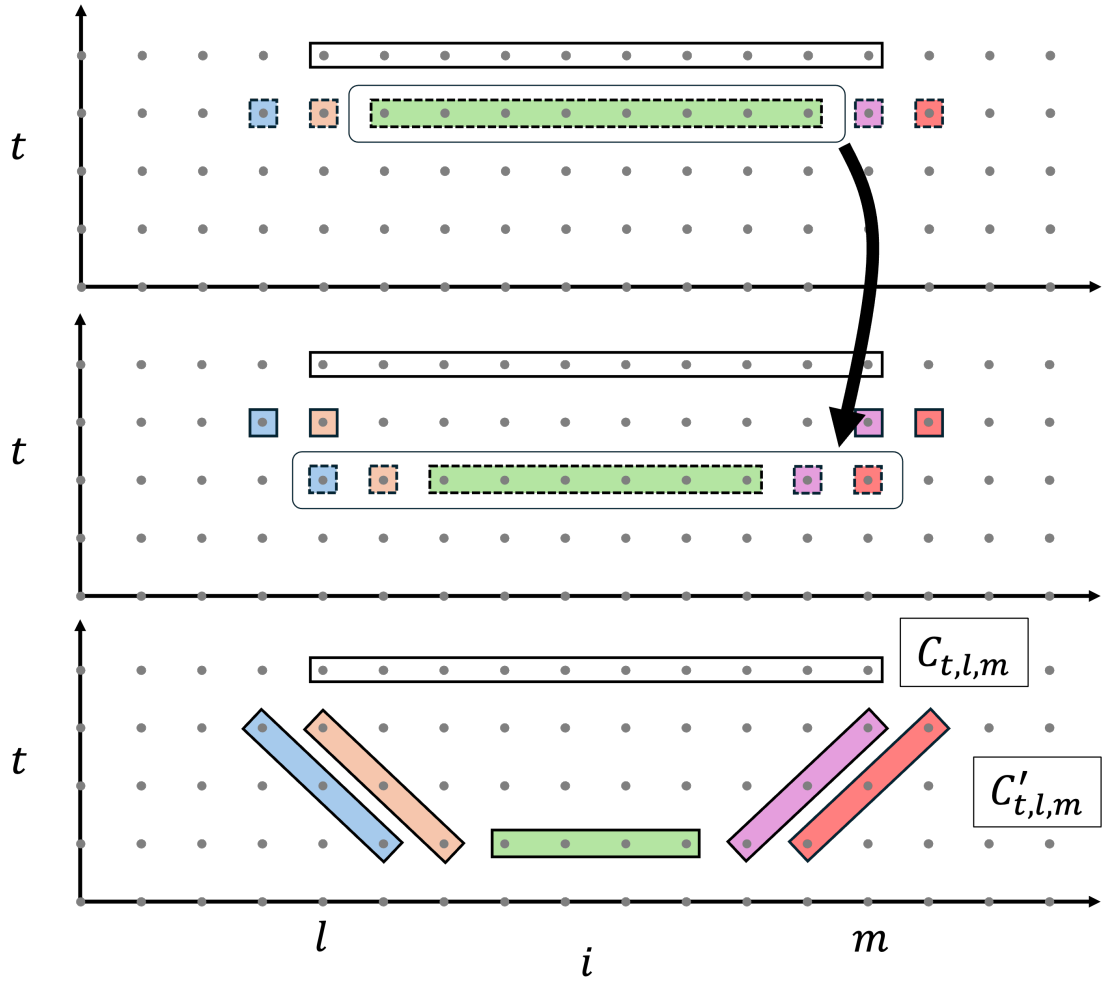


Figure 7.2: Checksum domains C and C' obtained from *partial* repeated substition.

We will call this scheme, *partial inlining*, in contrast to the *complete inlining* scheme in the previous section. Here, we obtain this recursive definition by only inlining parts of the new points introduced with each successive substitution (i.e., not every occurrence of the stencil variable on the RHS). This scheme differs from the previous *complete inlining* scheme. In order to evaluate C', we must read points on the bottom and oblique edges of these hyper-trapezoidal regions. The regions are still hyper-trapezoidal, albeit with a different shape. The control overhead of this scheme should be higher than that of the previous scheme because the values read at each point on the oblique edges are different at each time step. However, this scheme should require substantially less memory. We only need to precompute

and store five different weight combinations (one for each of the five colorings). Contrast this with the previous scheme, which requires precomputing and storing $l+m+\tau$ combinations of weights, one for each point in the trapezoidal base in Figure 7.1. Like the previous *complete inlining scheme*, we perform this partial inlining along each spatial dimension, so for a d-dimensional stencil, the checksum regions are hyper-trapezoidal. Next, we will consider an approach where we perform a variant of this partial inlining but only along one dimension.

7.3 New Checksum Scheme - Planar Collections

Finally, let us consider yet another way to construct C' by extending the ABFT scheme proposed by Cavelan and Ciorba [82]. We will start from the same definition of C' in Equation 7.5 that we obtained from a single substitution. In the previous section, we recognized that the inner summation term was actually $C'_{t-1,l+1,m-1}$. Expressing $C'_{t,l,m}$ as a function of $C'_{t-1,l+1,m-1}$ is what lead to the oblique boundaries illustrated in Figure 7.2. What if instead we express $C'_{t,l,m}$ as a function of $C'_{t-1,l,m}$, where the l and m index values are the same? This can be done by recognizing that the second, third, and fourth terms of Equation 7.6,

$$(w_0 + w_1)B_{t-1,l} + \left((w_0 + w_1 + w_2)\sum_{i=l+1}^{m-1} B_{t-1,i}\right) + (w_1 + w_2)B_{t-1,m}, \tag{7.9}$$

may be expressed as,

$$-(w_2)B_{t-1,l} + \left((w_0 + w_1 + w_2)\sum_{i=l}^{m} B_{t-1,i}\right) - (w_0)B_{t-1,m}, \tag{7.10}$$

Putting everything together allows us to express $C'_{t,l,m}$ as,

$$\begin{aligned} C'_{t,l,m} = {} & (w_0)B_{t-1,l-1} - (w_2)B_{t-1,l} \\ & + (w_0 + w_1 + w_2)C'_{t-1,l,m} \\ & - (w_0)B_{t-1,m} + (w_2)B_{t-1,m+1} \end{aligned} \tag{7.11}$$

Like the previous *partial inlining* scheme, after repeatedly inling Equation 7.11 τ times, we obtain this final recursive defintion,

$$\begin{aligned} C'_{t,l,m} = \; & (w_0) \sum_{0 \le p < \tau} \left(B_{t-1-p,l-1} - B_{t-1-p,m} \right) \\ & + (w_0 + w_1 + w_2)^{\tau} C'_{t-1,l,m} \\ & + (w_2) \sum_{0 \le p < \tau} \left(B_{t-1-p,m+1} - B_{t-1-p,l} \right) \end{aligned} \tag{7.12}$$

Recursively expressing $C'_{t,l,m}$ as a function of $C'_{t-1,l,m}$ has the effect of creating nice rectangular boundaries in the checksum domains as illustrated in Figure 7.3, as opposed to the oblique boundaries we saw in the previous two schemes. Unlike the other two *complete inlining* and *partial inlining* schemes, we only perform this scheme along a single dimension. This means we will have a series of independent trapezoidal regions. This is a ***critical*** differentiating property of this scheme. As such we will refer to this scheme as *rectangular* ***planar*** *inlining* to denote that we are dealing with collections of 2D rectangular slices in a (generally) d-dimensional domain.

7.4 Evaluation

We now have three alternate ways to carry out ABFT in stencils: *complete inlining* in all spatial dimensions (Section 7.1), *partial inlining* in all spatial dimensions (Section 7.2), and *rectangular planar inlining* in one spatial dimension (Section 7.3). We have implemented all three as a series of passes in the AlphaZ toolchain, augmenting an input stencil program with additional variables and equations to compute ABFT checksums, building on our prior automation work. This effectively involves carrying out the algebra described above to insert and manipulate the equations in the Alpha intermediate representation appropriately. Our toolchain takes a stencil Alpha program as input and ABFT size parameters to specify the size of the respective checksum regions (i.e., the sizes of the hyper-trapezoidal regions for the *complete* and *partial inlinging* schemes and the size of the rectangular regions for the

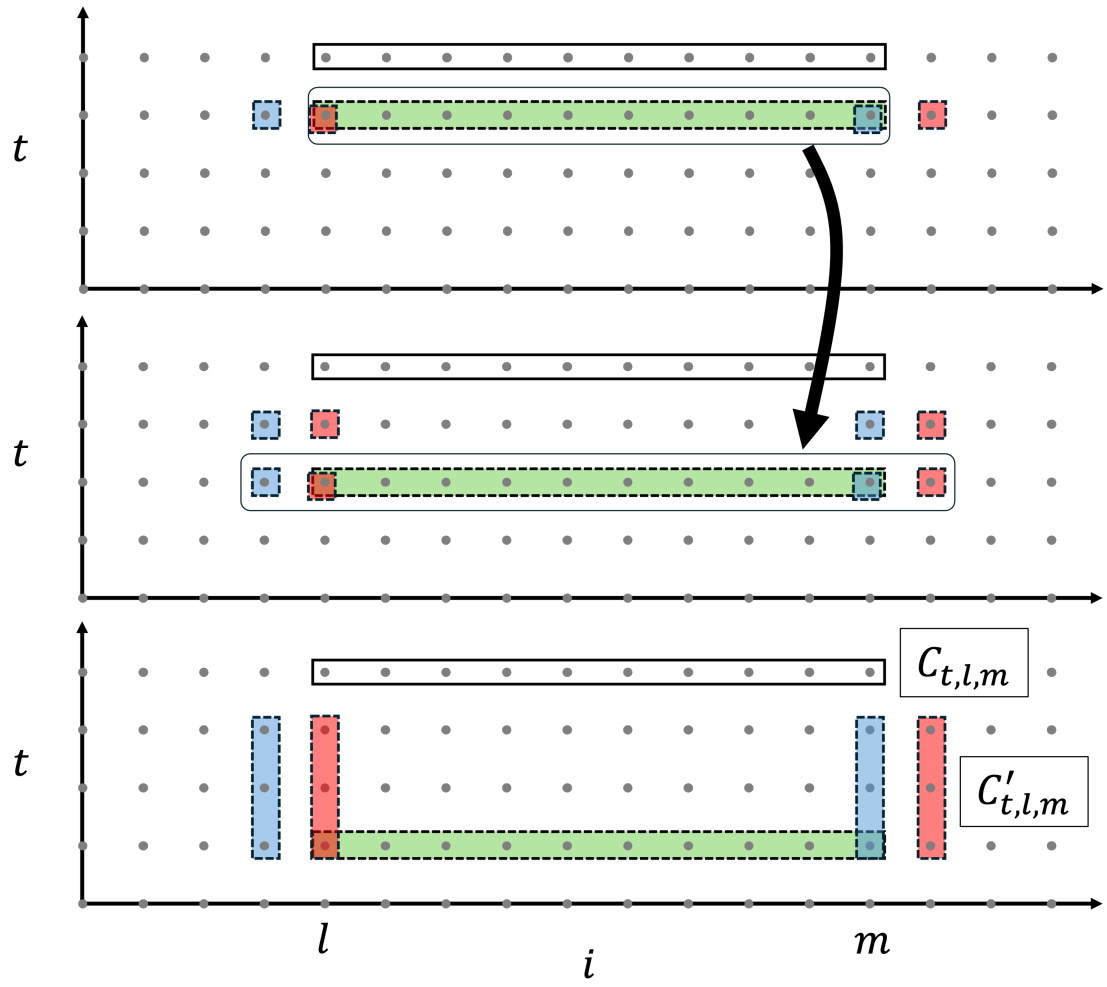


Figure 7.3: Rectangular domains C and C' obtained from *partial* repeated substition.

rectangular planar inlinging scheme). It generates four Alpha programs as output along with memory allocation and execution strategies (i.e., schedules) to execute each output program. The first output program is simply the input program and is used as our baseline. We have one additional output program for each of the three ABFT schemes. Each of these contains the same equations as the input program (i.e., the equations defining the primary stencil variable) and the additional equations implementing the corresponding ABFT checksums. Given the same input, each of the four programs computes the same stencil computation and produces identical outputs, in the absence of any errors.

7.4.1 Experimental Setup

We evaluate the three ABFT schemes discussed above on the following Alpha programs defining a 1D, 2D, and 3D stencil shown in Figures 7.4, 7.5, and 7.6 respectively. To meaningfully compare our three ABFT schemes we instrument a test harness to measure the following aspects of each ABFT variant:

- **Relative complexity** - the ratio of the number of operations required to execute the ABFT-augmented program to the input baseline program.
- **Execution time overhead** - the ratio of the run time of the ABFT-augmented program to the run time of the input program.
- **Error detection rate** - the percentage of time we could accurately measure checksum differences larger than some threshold (i.e., larger than the noise due to floating-point round-off errors) in the presence of random error injections.

```
affine star1d1r [T,N]->{: N>=20 and T>=10}
  inputs
    X : {[i] : 0<=i<=N}
  outputs
    Y : {[t,i] : 0<=t<=T and 0<=i<=N}
  let
  Y = case {
    {[t,i]: 0<=t<=1} : X[i];
    {[t,i]: 1<t and (i=0,N)} : Y[t-1,i];
    {[t,i]: 1<t and 0<i<N} :  0.3332[] * Y[t-1,i-1] +
                              0.3333[] * Y[t-1,i] +
                              0.33[] * Y[t-1,i+1];
  };
.
```

Figure 7.4: 1D stencil with constant weights.

Our toolchain generates four Alpha programs for each of these input programs, as described above, along with valid memory allocations and schedules. Note that we use a memory allocation that only maintains a history of as many copies of the stencil data needed per the program dependencies. In our examples, it is only necessary to maintain two data

```
affine star2d1r [T,N]->{: N>=20 and T>=10}
  inputs
    X : {[i,j] : 0<=i,j<=N}
  outputs
    Y : {[t,i,j] : 0<=t<=T and 0<=i,j<=N}
  let
    Y = case {
      {[t,i,j]: 0<=t<=1} : X[i,j];
      {[t,i,j]: 1<t and (i=0,N or j=0,N)} : Y[t-1,i,j];
      {[t,i,j]: 1<t and 0<i,j<N} : 0.5002[] * Y[t-1,i,j] +
        0.1247[] * Y[t-1,i-1,j] + 0.1249[] * Y[t-1,i+1,j] +
        0.1250[] * Y[t-1,i,j-1] + 0.1251[] * Y[t-1,i,j+1];
    };
.
```

Figure 7.5: 2D stencil with constant weights.

```
affine star3d1r [T,N]->{: N>=20 and T>=10}
  inputs
    X : {[i,j,k] : 0<=i,j,k<=N};
  outputs
    Y : {[t,i,j,k] : 0<=t<=T and 0<=i,j,k<=N}
  let
    Y = case {
      {[t,i,j,k]: 0<=t<=1} : X[i,j,k];
      {[t,i,j,k]: 1<t and (i=0,N or j=0,N or k=0,N)} : Y[t-1,i,j,k];
      {[t,i,j,k]: 1<t and 0<i,j,k<N} : 0.2500[] * Y[t-1,i,j,k] +
        0.1248[] * Y[t-1,i-1,j,k] + 0.1249[] * Y[t-1,i+1,j,k] +
        0.1250[] * Y[t-1,i,j-1,k] + 0.1250[] * Y[t-1,i,j+1,k] +
        0.1251[] * Y[t-1,i,j,k-1] + 0.1252[] * Y[t-1,i,j,k+1];
      };
.
```

Figure 7.6: 3D stencil with constant weights.

arrays when the stencil involves reading only points from the previous timestep and then swapping pointers—this corresponds to something like "$t\%2$" (modulo 2) accessing for the time dimension in the stencil variables. This memory mapping is needed minimally in order to run large problem sizes on any reasonable machine. Also, we do not consider parallelization or any complex tiling schemes here. While these are interesting, in this work, we only seek to validate whether or not a particular ABFT scheme is efficient in terms of the number of operations. If we expect the ABFT-augmented program to take more than twice as long as the original input program, for example, then we should not bother trying to find

good parallelizations or schedules anyway because we would never use this program in the first place. Therefore, we find the simplest sequential schedule that respects the memory mapping which corresponds to the following. At a particular time step, we first update the main stencil variable, then accumulate any points contributing to checksums at that same time step while their values are still resident in memory.

From this, we use a code generation strategy in Alpha to produce a single executable program with test harnessing used to measure the metrics described above. The test harness instrumentation does the following. First, we initialize the input stencil data array (e.g., the A variable in Equation 7.1) with random values. We allocate four copies of the stencil output array (e.g., the B variable in Equation 7.1) using 32-bit floating point data types, one for the input program and the other three for each of the three ABFT schemes. Then, we run each of the four programs with the same input data. For complexity and execution time, we report a single scalar value.

We measure the error detection rate *per bit*. We perform a single bit flip on a random element in the stencil data array at a random iteration space point and record the difference of checksum expressions surrounding the error injection site. If the difference is sufficiently large, then we signal an "error detection" event. We repeat this 500 times for each bit position from 31 down to 8 and report the percentage of time error detections were signaled for each bit position. We don't report anything on the lowest-order bits (0-7) because errors induced by flipping these bits are almost always too small to be observable.

We dynamically determine the threshold at which errors should be signaled by running the stencil computation for a small number of time steps (e.g., $T = 2H$ to guarantee that enough time steps have passed to allow the computation of at least one "full layer" of checksums) and computing the maximum value of all checksum difference expressions. Then, we set the threshold ten times larger than this max difference. For 32-bit floating point data types, in all of our experiments, this value ranges between 10^{-7} and 10^{-5}. Note that the noise characterizing the threshold is a property of the computation, and since each ABFT

Input program	H	L	T_{overhead}	N_{overhead}	T_{inj}	N_{inj}
star1d1r	[2..16]	[50..1000..50]	1000	4×10^6	$4H$	$3L$
star2d1r	[2..16]	[50..500..50]	250	5000	$4H$	$3L$
star3d1r	[2..16]	[10..100..5]	100	500	$4H$	$3L$

Table 7.1: Checksum size configurations explored for overhead and error injection experiments. H denotes the checksum region height along the time dimension (i.e., corresponding to the number of repeated substitutions). L denotes the side length along the space dimension. The notation "[a..b..c]" denotes the inclusive range of values between "a" and "b" strided by "c". Overhead experiments were run with problem sizes specified by T_{overhead} and T_{overhead} and the error injection experiments with T_{inj} and T_{inj}.

scheme computes checksums in slightly different ways, this enables us to set the threshold individually for each of the three schemes. All experiments are run on Linux machines with an Intel(R) Xeon(R) CPU E5-1650 v4 processor running at 3.60GHz, with 16 GB of memory. We report the range of checksum region configurations explored in Table 7.1. A single executable is generated for each configuration.

7.4.2 Overhead

Since Alpha maintains closed-form representations of the domains of all program variables, we can compute closed-form expressions characterizing the number of operations required to compute all points in all program variables. We estimate the overhead of a particular configuration (i.e., the particular ABFT scheme and the particular values of H and L) as the ratio of the complexities of the ABFT-augmented program to the input program. We compute the *complexity* of an Alpha program by counting the number of points in all context domains of all program equations as a function of the program size parameters T and N (not to be confused with the checksum size parameters H and L). We use the Barvinok counting library [57] to do this, which provides a polynomial over parameters given an input parameterized domain. If a program variable is computed using a reduction, then we count the context domain (Section 3.1.3) of the reduction body. The complexities of each scheme are reported as the surface plots in Figures 7.7, 7.8, and 7.9. The coloring shown in the figures highlight the exteme values. Points with the lowest complexities, when projected

down onto the bottom plane, are shown as dark blue. Those with higher complexities, when projected back onto the vertical planes, are shown as darker red. The corresponding relative measured execution times are shown as the solid points.

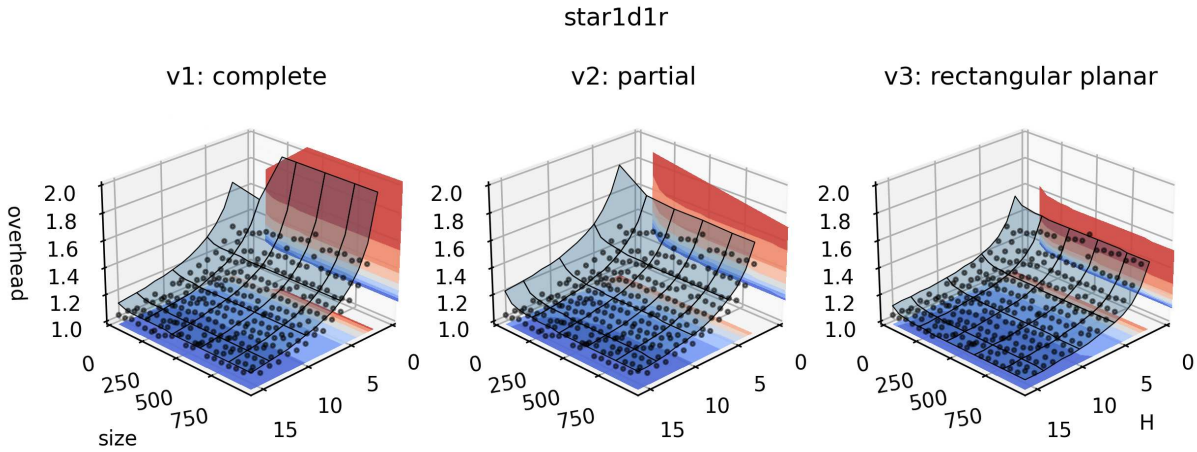


Figure 7.7: Complexities of the 1D stencil. The *size* axis shows the number of points given L. When there is only one spatial dimension, the sizes of each checksum for the complete and partial inlining schemes are true 2D trapezoids and, therefore, scales like the 2D rectangles of the rectangular planar inlining scheme.

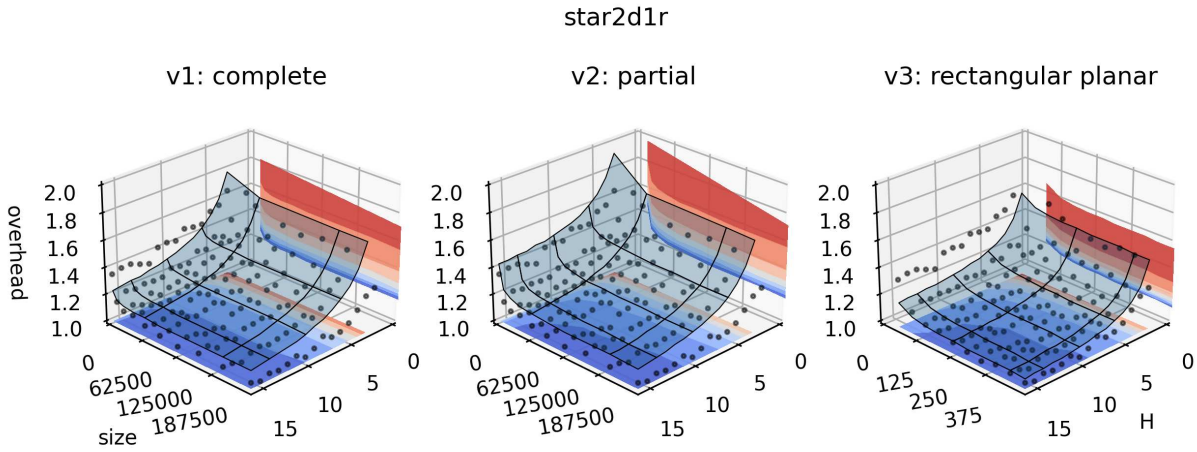


Figure 7.8: Complexities of the 2D stencil. Here, the sizes of the complete and partial inlining schemes are quadratic in L.

From this, we can see that for large values of L, the complexity and, thus, the overhead is primarily influenced by H. This agrees with prior work [82, 92], and is easy to understand

because larger H results in accumulating values less frequently. Conversely, when L becomes very small, the complexity shoots up sharply. This happens for the complete and partial inlining schemes because the hyper-trapezoids are *overlapped*. When $L \gg H$, adjacent trapezoids overlap minimally. When $L \ll H$, trapezoids degenerate into triangles, and adjacent triangles overlap almost entirely. This is exacerbated in higher dimensions, which is why we see higher spikes in the 2D and 3D plots. Configurations with overheads higher than two are not shown. Across all versions for the 1D, 2D, and 3D stencils, there are regions with low overhead, when both H and L are large, shown by the dark blue shadows on the bottom plane in the plots. Although, notice that the low overhead regions for the 2D and 3D stencils require checksums with very large sizes.

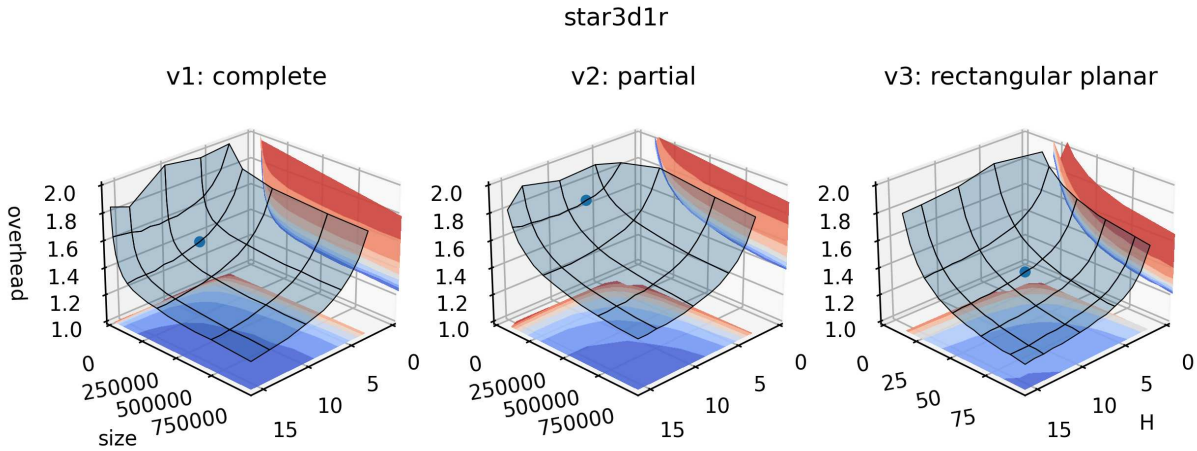


Figure 7.9: Complexities of the 3D stencil. Similarly, the sizes of the complete and partial inlining schemes are cubic in L. For example, the point $(H, L) = (8, 50)$ is shown explicitly and corresponds to a size of $2(50^3) = 250,000$, because computing checksum differences involves two accumulations of 50^3 points.

7.4.3 True errors

Regarding error detection, we expect the ABFT region size to influence our ability to observe non-negligible differences in checksums indicative of silent errors. When more points contribute to a particular checksum, an error will likely go undetected. However, we must be careful what we treat as a *true* error because some discrepancy is always present due to

floating-point round-off errors. A true error is one that causes the output to diverge enough that it is worth trying to catch. Suppose in our error injection experimentation, we induce a bit flip on one of the lowest order bits. We will likely not be able to detect this by looking at the difference in the checksum expression values surrounding the injection site because the effect of flipping the lowest-order bit is not likely to propagate into the accumulation involved. In isolation and of itself, we would incorrectly conclude that we missed detecting this error when we should have detcted it. However, the final values of the output stencil variable are equally likely to remain the same. Therefore, we only consider a particular bit flip as a *true* error if it causes the final answer to diverge significantly above the noise due to floating point round-off errors. To measure this, our instrumentation additionally reports the maximum difference between the error-injected stencil output and the baseline stencil output. If the enclosing checksum difference values are less than or equal to the floating-point round-off noise *and* every element in the output array is equivalent to the baseline output within floating-point round-off noise, then we do not classify this injection as a true error. By tuning the checksum region size, we expect to be able to find configurations that enable reasonable *true* error detection rates.

7.4.4 Single Error Injection

We report error injection detection rates per bit, as described previously. To be explicit, we run the baseline stencil program for T timesteps, do not perform error injection, and save its output. Then, we rerun the computation with the *same* input data, randomly flip bit "x" somewhere, and record the checksum difference values. Table 7.2 illustrates the error injection experimentation output for one particular configuration of the 3D stencil, for $H = 8$ and $L = 50$. Each value in the "v1 rate," "v2 rate," and "v3 rate" columns represents the percentage of errors signaled over 500 runs for each bit flip position.

The first row, for example, should be understood as follows. A detection rate of 0.00 indicates that after running the stencil computation 500 times, each time flipping bit 31

bit	v1 rate	v1 max error	v2 rate	v2 max error	v3 rate	v3 max error
31	0.00	1.721E-02	0.00	1.672E-02	100.00	3.759E-02
30	0.00	8.608E-03	0.00	8.361E-03	100.00	1.885E-02
29	100.00	1.588E+17	100.00	1.542E+17	100.00	3.455E+17
28	100.00	3.697E+07	100.00	3.591E+07	100.00	8.062E+07
27	0.00	8.608E-03	0.00	8.361E-03	100.00	1.871E-02
26	0.00	8.575E-03	0.00	8.328E-03	100.00	1.868E-02
25	0.00	8.070E-03	0.00	7.838E-03	100.00	1.762E-02
24	0.00	2.582E-02	0.00	2.508E-02	100.00	5.651E-02
23	0.00	4.304E-03	0.00	8.361E-03	100.00	1.852E-02
22	0.00	4.248E-03	0.00	2.162E-03	100.00	9.244E-03
21	0.00	2.124E-03	0.00	1.081E-03	100.00	4.651E-03
20	0.00	1.062E-03	0.00	5.406E-04	100.00	2.280E-03
19	0.00	5.310E-04	0.00	2.702E-04	100.00	1.150E-03
18	0.00	2.654E-04	0.00	1.351E-04	100.00	5.718E-04
17	0.00	1.327E-04	0.00	6.759E-05	100.00	2.864E-04
16	0.00	6.633E-05	0.00	3.379E-05	100.00	1.435E-04
15	0.00	3.314E-05	0.00	1.692E-05	100.00	7.146E-05
14	0.00	1.657E-05	0.00	8.523E-06	76.00	3.600E-05
13	0.00	8.285E-06	0.00	4.291E-06	35.00	1.800E-05
12	0.00	4.172E-06	0.00	2.145E-06	8.00	9.059E-06
11	0.00	2.145E-06	0.00	1.072E-06	0.00	4.529E-06
10	0.00	1.072E-06	0.00	5.364E-07	0.00	2.324E-06
9	0.00	4.768E-07	0.00	3.576E-07	0.00	1.192E-06
8	0.00	3.576E-07	0.00	1.192E-07	0.00	5.960E-07

Table 7.2: Example detection rates for the three different ABFT schemes displayed per bit for the 3D stencil with $H = 8$ and $L = 50$. The dynamically determined thresholds for each of the three ABFT schemes for this configuration run were 5.41^{-4}, 1.11^{-4}, and 3.04^{-6}, respectively. Using this, we can approximate floating-point round-off error noise near 10^{-6}. True undetected errors are shown in red, where the maximum relative error resulting from all bit flips is greater than the noise, 10^{-6}.

at a random iteration point, the signaling checksum difference for the complete inlining scheme (i.e., v1) was always indistinguishable from floating-point noise. Similarly, across all 500 runs, flipping bit 31 resulted in a maximum error in the output data array of 1.71^{-2}, which was at least two orders of magnitude larger than the checksum differences. Since the max error was larger than the floating-point noise, we count these misses as true undetected errors. True undetected errors are colored red in Table 7.2, true detected errors in green, and everything in blue. In other words, across all schemes, we can never detect errors injected by flipping bit position eight because the difference in the output is also indistinguishable from floating-point noise. We don't count points in blue as penalties in our evaluation.

We report the detection rate for the 1D, 2D, and 3D stencils across each of the three ABFT schemes in Figures 7.10, 7.11, and 7.12 respectively. Each figure plots the error detection rate for each of the three ABFT schemes bucketed by bit flip position. For the 1D

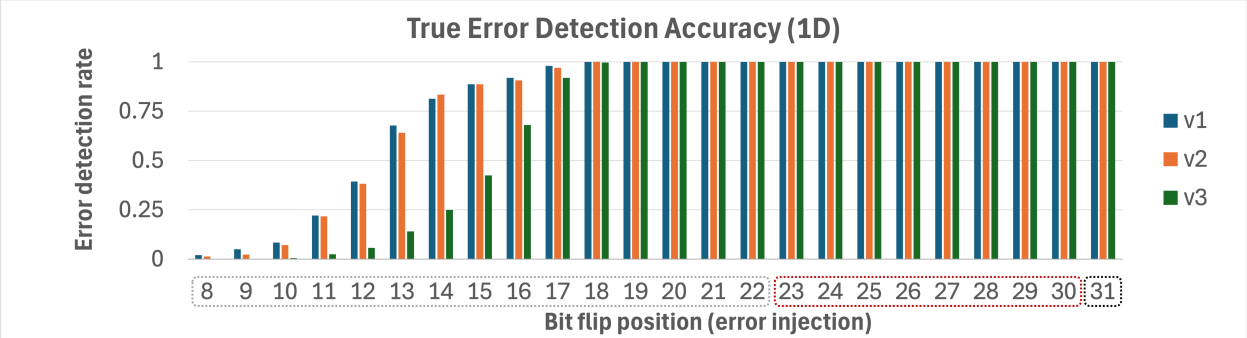


Figure 7.10: Error detection rate for the 1D stencil across all checksum size configurations, bucketed by bit flip position.

stencil, all three ABFT schemes were always able to detect errors injected in the higher-order bits. Below bit 18, the complete and partial inlining schemes performed better.

However, for the 2D and 3D cases, we can see that the effectiveness of the complete and partial inlining schemes drops substantially. The reason for this is that among all the 2D and 3D configurations explored, most of them involve checksums computed using simply too many points, which swallows errors, making their presence indistinguishable from floating point noise. The particular instance from Table 7.2 corresponds to $H = 8$ and $L = 50$,

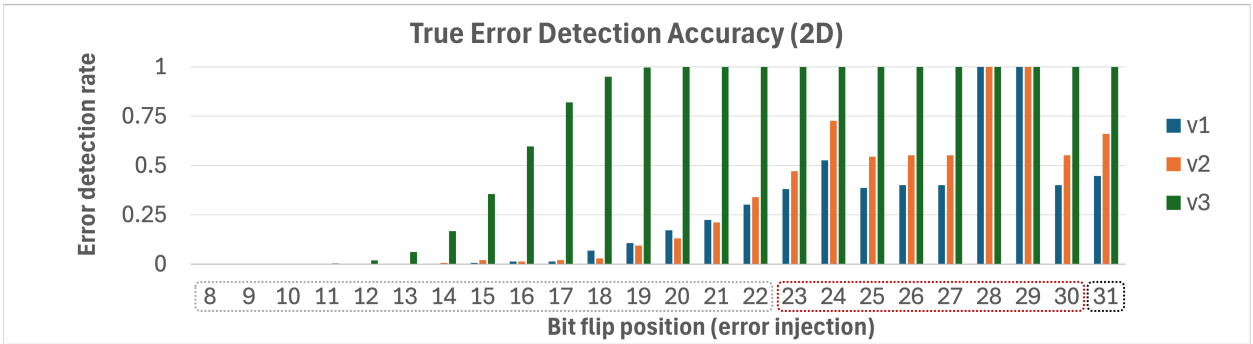


Figure 7.11: Error detection rate for the 2D stencil across all checksum size configurations, bucketed by bit flip position.

shown by the single point, overlaid on the plot in Figure 7.9, has a size of 250,000 (i.e., each checksum difference involves on the order of 10^5 accumulations). Additionally, this is even among the configurations with the lowest overhead, which involve even larger sizes. In retrospect, this makes sense because we get low overhead by making the hyper-trapezoidal regions large in H and L. Contrast this with the planar rectangular inlining scheme, whose size is linear in L even for the 2D and 3D schemes. Since this effectively involves a series of 2D slices, it results in the same detection effectiveness as the 1D stencil instance because it involves substantially fewer points than the configurations with the same overhead for the other schemes.

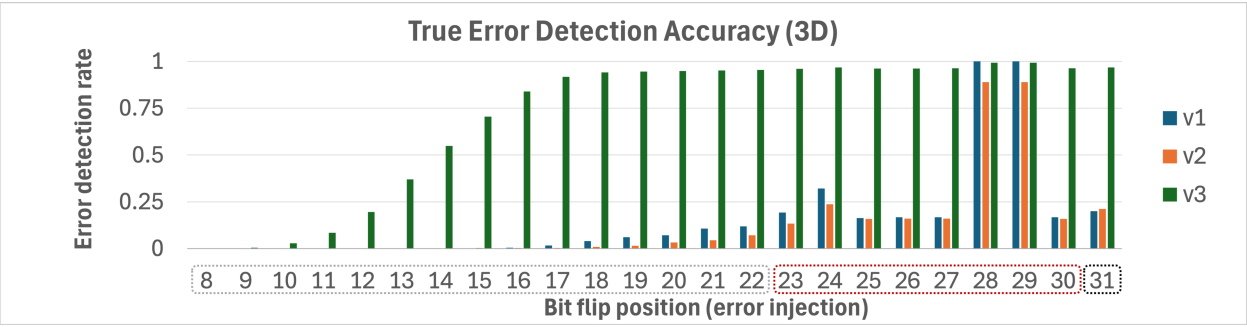


Figure 7.12: Error detection rate for the 3D stencil across all checksum size configurations, bucketed by bit flip position.

7.5 Conclusions

As we have discussed extensively, transient silent data corruption errors are difficult to deal with because they manifest as errors in the computed data. Detecting them, therefore, requires running some analysis on the computed data, typically in the form of clever algebra. This is challenging because it involves a very manual analysis of the particular target program at hand, and historically, this has only been done manually. Since Alpha is itself an equational language, it is aptly suited for the types of analysis required to programmatically and systematically carry out such algebraic transformations. In the previous chapter, we proposed automation strategies for carrying out ABFT analysis for stencil computations in Alpha. In this chapter, we have discussed two additional ABFT checksum schemes that can be used to realize the same goal with varying levels of effectiveness. In doing so, we provided a thorough experimental analysis of the tradeoff between the two aspects of performance characterizing ABFT: overhead and error detection rate. We need small sizes to obtain high detection rates. We also need large sizes to obtain low overheads. However, as we saw in the higher-dimensional stencils, many of the configurations we explored had difficulty satisfying both objectives. They were either too large to reliably detect most errors or too small that the overhead incurred was too large to be worth it.

By leveraging the prior work of Cavelan and Ciorba [82] into our automation framework to carry out the planar rectangular inlining scheme, we showed how to obtain configurations that have both a low overhead and a high detection rate. It was possible to carry out this scheme in only one of the spatial dimensions because the recursive definition of the checksum, from Equation 7.11, is rectangular. The checksum, C', could be expressed as a function of another checksum instance at the previous time step with the *same size*, which allows terms along the other dimensions to be cleanly factored out. It may be possible to extend the complete and partial inlining such that we only expand along a single dimension, but doing so would require us to figure out how to perform the analogous factorization of checksum

instances when the checksum instance size (i.e., $C'_{t-1,l-1,m+1}$ $C'_{t-1,l+1,m-1}$) differs at each step. We are not sure if this is possible at this point, but if we could do so, then perhaps we would improve detection rates even further since the first two schemes performed better for the 1D stencil. Additionally, we have not considered methods for reasoning about how to navigate the multi-objective search space. These are interesting observations that could inform future work.

Chapter 8

Future Work and Conclusions

In this dissertation, we have talked extensively about program optimizations involving reductions. We focused on two ostensibly separate and unrelated problems—the simplifying reductions transformation and the automation of Algorithm Based Fault Tolerance (ABFT) with a focus on a particular class of scientific computing codes known as stencil computations. At this point we have everything needed in order motivate the deeper connection between these two ideas. The main idea behind ABFT is to augment the computation with extra work using invariant checksums (i.e., reductions) by exploiting algebraic identities, which can be used to signal error when their difference rises above some threshold. The key observation is that when the target application itself involves reductions of some sort, the application of ABFT inherently involves computing reductions with precisely the kind of redundancy amenable to simplification. Let us conclude by illustrating how we can leverage simplification and the automation proposed in this work to automatically rediscover the original 1984 ABFT result of Huang and Abraham.

8.1 ABFT from the Lens of Simplification

Recall the classic matrix multiplication for square N-by-N matrices where $C = A \times B$, specified by the equation:

$$C_{i,j} = \sum_{k=1}^{N} \left(A_{i,k} \times B_{k,j} \right) \tag{8.1}$$

Huang and Abraham defined β_i, γ_i and γ'_i as, respectively, the row checksums of B and C, and the inner product of a row of A and the checksum vector β as defined in Equations 8.2 and 8.3 (and illustrated in Figure 8.1).

$$\gamma_i = \sum_{j=1}^{N} C_{i,j} \tag{8.2}$$

$$\gamma'_i = \sum_{j=1}^{N} \left(A_{i,j} \times \beta_j\right), \qquad \beta_i = \sum_{j=1}^{N} B_{i,j} \tag{8.3}$$

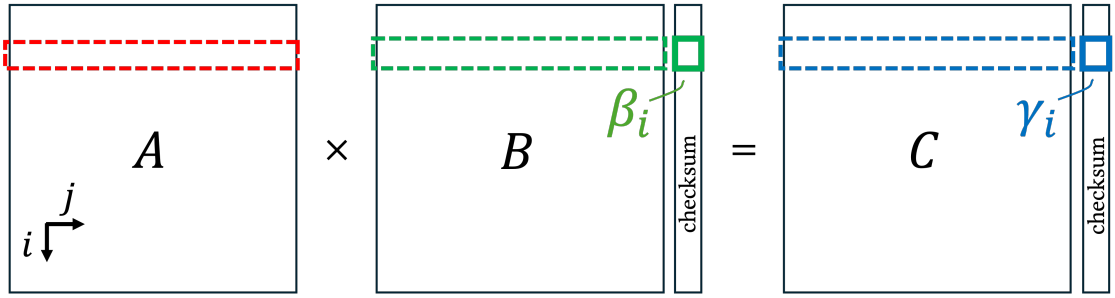


Figure 8.1: ABFT checksums, (from Huang and Abraham [7]).

The two quantities γ_i and γ'_i denote the checksums that compute the same numerical value, and errors occurring during the computation of Equation 8.1 can be detected when their difference, $|\gamma_i - \gamma'_i|$, is sufficiently large. More importantly, the $O(N^2)$ cost of computing Equations 8.2 and 8.3 is *cheap* relative to the main $O(N^3)$ computation in Equation 8.1. The key insight here is that the task of constructing ABFT checksums can be cast as an instance of the simplifying reductions problem. Consequentially, cheap checksum specifications can be obtained systematically from simplification.

To understand how the same effect can be achieved with simplification, let us start by defining the row checksums, γ_i, over the output, C, as the following:

$$\gamma_i = \sum_{j=0}^{N-1} C_{i,j} \tag{8.4}$$

Substituting $C_{i,j}$ with its definition from Equation 8.1 lets us express γ_i as:

$$\begin{gathered} \gamma_i = \sum_{\{j,k\} \in \mathcal{D}} \left(A_{i,k} \times B_{k,j}\right) \\ \mathcal{D} = \{[j,k] \mid (0 \leq j < N) \wedge (0 \leq k < N)\} \end{gathered} \tag{8.5}$$

As written, the computation specified by Equation 8.5 is cubic, $O(N^3)$, but it can be systematically rewritten with quadratic, $O(N^2)$, complexity via simplification.

Simplification discovers the following three things about Equation 8.5. First, values are accumulated over a 2D space (over j and k). Second, the multiplication operator inside the reduction body distributes over the reduction addition operator. Third and finally, the subexpression $A_{i,k}$ evaluates to the same value along one of the accumulation dimensions (i.e., $A_{i,k}$ is independent of j), and therefore the resuse space is non-empty. Putting everything together allows the problem to be decomposed into a reduction of reductions, such that the outer reduction accumulates over k and the inner reduction over j.

$$\gamma_i = \sum_{k=0}^{N-1} \sum_{j=0}^{N-1} \left(A_{i,k} \times B_{k,j} \right) \tag{8.6}$$

Then, the term $A_{i,k}$ may be factored out of the inner reduction, and the inner summation over j can be separated into its own equation.

$$\gamma_i = \sum_{k=0}^{N-1} \left(A_{i,k} \times X_k \right) \qquad\qquad X_k = \sum_{j=0}^{N-1} B_{k,j} \tag{8.7}$$

Finally, these equations together give the $O(N^2)$ complexity version of the same computation specified by Equation 8.4. The fact that summing rows of B is required to detect errors in a particular row of the output C falls out systematically from the simplification of Equation 8.5.

Complementarily, we would want to perform a similar simplification of checksums over *columns* of C to pinpoint the error's location and fix it in the output. We don't show it here because it is analogous to the simplification described above. In the end, we end up with the original $O(N^3)$ matrix product and four $O(N^2)$ checksums (two pairs of checksums over the columns and rows of the inputs and outputs). This is precisely what Huang and Abraham [7] propose.

8.2 Concluding remarks

In conclusion, we have discussed the optimization of programs containing reductions within the polyhedral compilation framework with respect to the reduction simplification transfor-

mation and the automation of Algorithm-Based Fault Tolerance (ABFT). We have addressed several of the theoretical gaps in the original simplification formulation, shown how to implement the first push-button simplification engine, and broadened its scope of applicability to a strictly broader class of programs than previously supported. In doing so, we have shown that simplification rediscovers several key results from RNA folding in bioinformatics algorithms and Algorithm-Based Fault Tolerance.

From the perspective of ABFT, we proposed several new automation techniques that can be used to carry out ABFT in a polyhedral compiler for a class of scientific codes called stencil computations. We provided an extensive analysis of the tradeoffs between design choices. We motivated that the ABFT problem should be viewed as a multi-objective problem, where we seek to minimize overhead while simultaneously maximizing the error detection rate. As we have shown, the two objects pull in opposite directions. At this time, we do not know the best way to explore this space. However, with the automation proposed in this work, we open the door for more comprehensive design-space exploration. This was impossible with prior work, which required very careful and painful manual analysis and implementation tailored specifically to the target input application at hand. By making a connection reduction simplification and the analysis representative of ABFT, we open the door for opportunities to do generalized automatic Algorithm-Based Fault Tolerance in arbitrary input programs.

The ultimate goal is to enable compilers to take a high-level application program specification and carry it out in the most efficient way possible, preferably *automatically* and *optimally*. This work takes a step in that direction to enable users (i.e., application programmers) to focus less on the *engineering* aspects (the *how*) of their algorithms and more on the problems (the *what*) that their algorithms are intended to solve.

Bibliography

[1] R. Lyons and W. Vanderkulk. "The Use of Triple-Modular Redundancy to Improve Computer Reliability". In: *IBM Journal of Research and Development* 6.2 (1962), pp. 200–209.

[2] Ruth Nussinov et al. "Algorithms for Loop Matchings". In: *SIAM Journal on Applied Mathematics* 35.1 (1978), pp. 68–82. DOI: 10.1137/0135006.

[3] R. Nussinov and A. Jacobson. "Fast algorithm for predicting the secondary structure of single stranded RNA". In: *Proc. Nat. Acad. Sci. USA* 77.11 (1980), pp. 6309–6313.

[4] Michael Zuker and Patrick Stiegler. "Optimal computer folding of large RNA sequences using thermodynamics and auxiliary information". In: *Nucleic Acids Research* 9.1 (1981), pp. 133–148.

[5] S. Ganapathy and T. G. Dennehy. "A new general triangulation method for planar contours". In: *SIGGRAPH Comput. Graph.* 16.3 (July 1982), pp. 69–75. ISSN: 0097-8930. DOI: 10.1145/965145.801264. URL: https://doi.org/10.1145/965145.801264 (visited on 06/27/2024).

[6] J. A. B. Fortes and D. I. Moldovan. "Data broadcasting in linearly scheduled array processors". In: *SIGARCH Comput. Archit. News* 12.3 (Jan. 1984), pp. 224–231. ISSN: 0163-5964. DOI: 10.1145/773453.808186. URL: https://dl.acm.org/doi/10.1145/773453.808186 (visited on 10/03/2024).

[7] Kuang-Hua Huang and Jacob A. Abraham. "Algorithm-Based Fault Tolerance for Matrix Operations". In: *IEEE Transactions on Computers* C-33.6 (June 1984). Number:

6 Conference Name: IEEE Transactions on Computers, pp. 518–528. ISSN: 1557-9956. DOI: 10.1109/TC.1984.1676475. URL: https://ieeexplore.ieee.org/abstract/document/1676475 (visited on 05/20/2024).

[8] Guo-Jie Li and Benjamin W. Wah. "Systolic Processing for Dynamic Programming Problems". In: *International Conference on Parallel Processing, ICPP'85*. IEEE Computer Society Press, 1985, pp. 434–441.

[9] 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, Dec. 1986, pp. 488–503.

[10] Michael Wolfe. "Iteration Space Tiling for Memory Hierarchies". In: *Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing*. USA: Society for Industrial and Applied Mathematics, Dec. 1987, pp. 357–361. ISBN: 978-0-89871-228-5. (Visited on 05/29/2024).

[11] F. Irigoin and R. Triolet. "Supernode Partitioning". In: *15th ACM Symposium on Principles of Programming Languages*. ACM, Jan. 1988, pp. 319–328.

[12] J. Pearl. *Probabilistic Reasoning in Intelligent Systems*. San Mateo, CA: Morgan Kaufmann, 1988.

[13] M. S. Lam. *A Systolic Array Optimizing Computer*. Kluwer Academic (Springer), 1989. DOI: 10.1007/978-1-4613-1705-0.

[14] Christophe Mauras. "Alpha : un langage equationnel pour la conception et la programmation d'architectures paralleles synchrones". These de doctorat. Rennes 1, Jan. 1989. URL: https://theses.fr/1989REN10116 (visited on 05/29/2024).

[15] P. Quinton and V. Van Dongen. "The Mapping of Linear Recurrence Equations on Regular Arrays". In: *Journal of VLSI Signal Processing* 1.2 (1989). Publisher: Kluwer Academic Publishers, Boston, pp. 95–113.

[16] S. V. Rajopadhye. “Synthesizing Systolic Arrays with Control Signals from Recurrence Equations”. In: *Distributed Computing* 3 (May 1989). Publisher: Elsevier Science, North Holland, pp. 88–105.

[17] J. Ramanujam and P. Sadayappan. “Nested Loop Tiling for Distributed Memory Machines”. In: *Proceedings of the Fifth Distributed Memory Computing Conference, 1990.* Vol. 2. Charleston, SC, USA: IEEE, Apr. 1990, pp. 1088–1096. ISBN: 0-8186-2113-3. DOI: 10.1109/DMCC.1990.556321. URL: https://ieeexplore.ieee.org/abstract/document/556321 (visited on 05/29/2024).

[18] Robert Schreiber and Jack J. Dongarra. *Automatic blocking of nested loops*. Tech. rep. NASA-CR-188874. Issue: NASA-CR-188874 NTRS Author Affiliations: Research Inst. for Advanced Computer Science, Tennessee Univ. NTRS Document ID: 19910023530 NTRS Research Center: Legacy CDMS (CDMS). RIACS, Aug. 1990. URL: https://ntrs.nasa.gov/citations/19910023530 (visited on 05/29/2024).

[19] G. R. Shafer and P. P. Shenoy. “Probability Propagation”. In: *Annals of Mathematics and Artificial Intelligence* 2.1-4 (Mar. 1990). Number: 1-4, pp. 327–351.

[20] P. Feautrier. “Dataflow analysis of array and scalar references”. In: *International Journal of Parallel Programming* 20.1 (Feb. 1991), pp. 23–53.

[21] H. Le Verge, C. Mauras, and P. Quinton. “The ALPHA Language and its use for the Design of Systolic Arrays”. In: *Journal of VLSI Signal Processing* 3.3 (Sept. 1991), pp. 173–182.

[22] William Pugh. “The Omega test: a fast and practical integer programming algorithm for dependence analysis”. In: *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*. Supercomputing ’91. New York, NY, USA: Association for Computing Machinery, Aug. 1991, pp. 4–13. ISBN: 978-0-89791-459-8. DOI: 10.1145/125826.125848. URL: https://dl.acm.org/doi/10.1145/125826.125848 (visited on 05/29/2024).

[23] M. Wolf and M. Lam. “Loop transformation theory and an algorithm to maximize parallelism”. In: *IEEE Transactions on Parallel and Distributed Systems* 2.4 (Oct. 1991). Number: 4, pp. 452–471.

[24] Michael E. Wolf and Monica S. Lam. “A data locality optimizing algorithm”. en. In: *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation - PLDI ’91*. Toronto, Ontario, Canada: ACM Press, 1991, pp. 30–44. ISBN: 978-0-89791-428-4. DOI: 10.1145/113445.113449. URL: http://portal.acm.org/citation.cfm?doid=113445.113449 (visited on 05/29/2024).

[25] Paul Feautrier. “Some efficient solutions to the affine scheduling problem. I. One-dimensional time”. en. In: *International Journal of Parallel Programming* 21.5 (Oct. 1992). Number: 5, pp. 313–347. ISSN: 1573-7640. DOI: 10.1007/BF01407835. URL: https://doi.org/10.1007/BF01407835 (visited on 05/09/2024).

[26] Paul Feautrier. “Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time”. en. In: *International Journal of Parallel Programming* 21.6 (Dec. 1992), pp. 389–420. ISSN: 1573-7640. DOI: 10.1007/BF01379404. URL: https://doi.org/10.1007/BF01379404 (visited on 10/03/2024).

[27] H. Le Verge. “Un environnement de transformations de programmmes pour la synthèse d’architectures régulières”. PhD Thesis. IRISA, Campus de Beaulieu, Rennes, France: L’Université de Rennes I, Oct. 1992.

[28] S. V. Rajopadhye, L. Mui, and S. Kiaei. “Piecewise Linear Schedules for Recurrence Equations”. In: *VLSI Signal Processing, V*. Ed. by K. Yao et al. Napa: IEEE Signal Processing Society, Oct. 1992, pp. 375–384.

[29] S. Amarasinghe and M. Lam. “Communication Optimization and Code Generation for Distributed Memory Machines”. In: *The ACM SIGPLAN ‘93 Conference on Programming Language Design and Implementation*. Albuquerque, N.M.: ACM Press, June 1993, pp. 126–138.

[30] C. Lengauer. “Loop Parallelization in the Polytope Model”. In: *CONCUR 93*. Ed. by E. Best. Springer–Verlag Lecture Notes in Computer Science. Issue: 715. 1993, pp. 398–416.

[31] Xavier Redon and Paul Feautrier. “Scheduling Reductions”. In: *Proceedings of the 8th International Conference on Supercomputing*. ICS ’94. event-place: Manchester, England. New York, NY, USA: ACM, 1994, pp. 117–125. ISBN: 0-89791-665-4. DOI: 10.1145/181181.181319. URL: http://doi.acm.org/10.1145/181181.181319.

[32] W. Sying-Jyan and N. Jha. “Algorithm-based fault tolerance for FFT networks”. In: *IEEE Transactions on Computers* 43.7 (1994), pp. 849–854.

[33] F. de Dinechin, P. Quinton, and T. Risset. “Structuration of the ALPHA language”. In: *Programming Models for Massively Parallel Computers*. Oct. 1995, pp. 18–24. DOI: 10.1109/PMMPC.1995.504337. URL: https://ieeexplore.ieee.org/abstract/document/504337?casa_token=6p-mQuhnNoIAAAAA:B1F4oLBCcBFnVpjSm865NB2uxLkjDsiO2qKEY-OfzfcIEGFA4Hg37JsJpGvfxOOp8uvb7Hdn6wo (visited on 09/30/2024).

[34] Atul Narkhede and Dinesh Manocha. “VII.5 - Fast Polygon Triangulation Based on Seidel’s Algorithm”. In: *Graphics Gems V*. Ed. by Alan W. Paeth. Boston: Academic Press, Jan. 1995, pp. 394–397. ISBN: 978-0-12-543457-7. DOI: 10.1016/B978-0-12-543457-7.50059-0. URL: https://www.sciencedirect.com/science/article/pii/B9780125434577500590 (visited on 06/27/2024).

[35] D. Wilde and S. Rajopadhye. “The naive execution of affine recurrence equations”. In: *Proceedings The International Conference on Application Specific Array Processors*. ISSN: 1063-6862. Strasbourg, France: IEEE, July 1995, pp. 1–12. DOI: 10.1109/ASAP.1995.522900. URL: https://ieeexplore.ieee.org/abstract/document/522900 (visited on 05/28/2024).

[36] A. Roy-Chowdhury, N. Bellas, and P. Banerjee. “Algorithm-based error-detection schemes for iterative solution of partial differential equations”. In: *IEEE Transactions on Computers* 45.4 (1996), pp. 394–407.

[37] Vincent Loechner and Doran K. Wilde. “Parameterized Polyhedra and Their Vertices”. In: *IJPP: International Journal of Parallel Programming* 25.6 (Dec. 1997). Publisher: Springer Verlag, pp. 525–549. DOI: 10.1023/A:1025117523902. URL: https://link.springer.com/article/10.1023/A:1025117523902.

[38] R. B. Lyngso, M. Zuker, and C. N. S. Pedersen. “Fast Evaluation of Internal Loops in RNA Secondary Structure Prediction”. In: *Bioinformatics* 15 (June 1999), pp. 440–445.

[39] S. M. Aji and R. J. McEliece. “The Generalized Distributive Law”. In: *IEEE Transactions on Information Theory* 46.2 (Mar. 2000). Number: 2, pp. 325–343.

[40] C. Flamm et al. “RNA folding at elementary step resolution”. In: *RNA* 6 (Mar. 2000), pp. 325–338.

[41] M. Griebl, P. Feautrier, and C. Lengauer. “Index Set Splitting”. In: *IJPP: International Journal of Parallel Programming* 28.6 (Dec. 2000), pp. 607–631. DOI: https://doi.org/10.1023/A:1007516818651.

[42] X. Redon and P. Feautrier. “Detection of Scans in the Polytope Model”. In: *Parallel Algorithms and Applications* 15.3-4 (2000). Number: 3-4, pp. 229–263.

[43] F. R. Kschischang, B. J. Frey, and H-A. Loeliger. “Factor Graphs and the Sum-Product Algorithm”. In: *IEEE Transactions on Information Theory* 47.2 (Feb. 2001). Number: 2, pp. 498–519.

[44] Robert M. Dirks and Niles A. Pierce. “A partition function algorithm for nucleic acid secondary structure including pseudoknots”. en. In: *Journal of Computational Chemistry* 24.13 (2003). Number: 13 _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/jcc.10296,

pp. 1664–1677. ISSN: 1096-987X. DOI: 10.1002/jcc.10296. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/jcc.10296 (visited on 05/27/2024).

[45] R. J. Drost and A. C. Singer. "Image Segmentation using Factor Graphs". In: *IEEE Workshop on Statistical Signal Processing*. Sept. 2003, pp. 150–153.

[46] P. Pakzad and V. Anantharam. "A new look at the generalized distributive law". In: *IEEE Transactions on Information Theory* 50.6 (June 2004). Number: 6, pp. 1132–1155. ISSN: 1557-9654. DOI: 10.1109/TIT.2004.828058.

[47] Michael T. Wolfinger et al. "Efficient computation of RNA folding dynamics". In: *Journal of Physics A: Mathematical and General* 37.17 (2004), pp. 4731–4741.

[48] G.A. Reis et al. "SWIFT: software implemented fault tolerance". In: *International Symposium on Code Generation and Optimization*. Mar. 2005, pp. 243–254. DOI: 10.1109/CGO.2005.34. URL: https://ieeexplore.ieee.org/abstract/document/1402092 (visited on 08/27/2024).

[49] Gautam and S. Rajopadhye. "Simplifying Reductions". In: *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '06. event-place: Charleston, South Carolina, USA. New York, NY, USA: Association for Computing Machinery, 2006, pp. 30–41. ISBN: 1-59593-027-2. DOI: 10.1145/1111037.1111041. URL: https://doi.org/10.1145/1111037.1111041.

[50] David H Mathews and Douglas H Turner. "Prediction of RNA secondary structure by free energy minimization". In: *Current Opinion in Structural Biology*. Nucleic acids/Sequences and topology 16.3 (June 2006). Number: 3, pp. 270–278. ISSN: 0959-440X. DOI: 10.1016/j.sbi.2006.05.010. URL: https://www.sciencedirect.com/science/article/pii/S0959440X06000819 (visited on 05/17/2024).

[51] M. Patra and M. Karttunen. "Stencils with isotropic discretization error for differential operators". In: *Numerical Methods for Partial Differential Equations* 22.4 (2006), pp. 936–953.

[52] N. Aggarwal et al. “Configurable isolation: Building high availability systems with commodity multi-core processors”. In: *SIGARCH Comput. Archit. News* 35.2 (2007). Place: New York, NY, USA Publisher: Association for Computing Machinery, pp. 470–481.

[53] R. J. Drost and A. C. Singer. “Factor-Graph Algorithms for Equalization”. In: *IEEE Transactions on Signal Processing, Part 2* 55.5 (May 2007). Number: 5, pp. 2052–2065.

[54] “Convex Polytopes”. en. In: *Convex and Discrete Geometry*. Ed. by Peter M. Gruber. Berlin, Heidelberg: Springer, 2007, pp. 243–351. ISBN: 978-3-540-71133-9. DOI: 10.1007/978-3-540-71133-9_3. URL: https://doi.org/10.1007/978-3-540-71133-9_3 (visited on 06/27/2024).

[55] A. Jagirdar, R. Oliveira, and T. Chakraborty. “Efficient flip-flop designs for SET/SEU mitigation with tolerance to crosstalk induced signal delays”. In: *Proc. IEEE Workshop Silicon Errors Logic Syst. Effects*. Citeseer, 2007.

[56] Nicolas Vasilache, Albert Cohen, and Louis-Noel Pouchet. “Automatic Correction of Loop Transformations”. In: *16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007)*. 2007, pp. 292–304. DOI: 10.1109/PACT.2007.4336220.

[57] Sven Verdoolaege et al. “Counting Integer Points in Parametric Polytopes Using Barvinok’s Rational Functions”. en. In: *Algorithmica* 48.1 (May 2007), pp. 37–66. ISSN: 1432-0541. DOI: 10.1007/s00453-006-1231-0. URL: https://doi.org/10.1007/s00453-006-1231-0 (visited on 10/03/2024).

[58] G. Bosilca et al. “Algorithm-based fault tolerance applied to high performance computing”. In: *Journal of Parallel and Distributed Computing* 69.4 (2009), pp. 410–416.

[59] Hamidreza Chitsaz, Rolf Backofen, and S.Cenk Sahinalp. “biRNA: Fast RNA-RNA Binding Sites Prediction”. In: *Workshop on Algorithms in Bioinformatics (WABI)*.

Ed. by S.L. Salzberg and T. Warnow. Vol. 5724. LNBI. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 25–36.

[60] R. G. Dreslinski et al. "Near-threshold computing: Reclaiming Moore's law through energy efficient integrated circuits". In: *Proceedings of the IEEE* 98.2 (2010), pp. 253–266.

[61] Arpith C. Jacob, Jeremy D. Buhler, and Roger D. Chamberlain. "Rapid RNA Folding: Analysis and Acceleration of the Zuker Recurrence". en. In: *2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*. Charlotte, NC, USA: IEEE, 2010, pp. 87–94. DOI: 10.1109/FCCM.2010.22. URL: http://ieeexplore.ieee.org/document/5474066/ (visited on 05/17/2024).

[62] Sven Verdoolaege. "isl: An Integer Set Library for the Polyhedral Model". en. In: *Mathematical Software – ICMS 2010*. Ed. by Komei Fukuda et al. Berlin, Heidelberg: Springer, 2010, pp. 299–302. ISBN: 978-3-642-15582-6. DOI: 10.1007/978-3-642-15582-6_49.

[63] Ronny Lorenz et al. "ViennaRNA Package 2.0". In: *Algorithms for molecular biology* 6.1 (2011). Publisher: Springer, pp. 1–14.

[64] Louis-Noël Pouchet et al. "Loop Transformations: Convexity, Pruning and Optimization". In: *SIGPLAN Not.* 46.1 (Jan. 2011). Place: New York, NY, USA Publisher: Association for Computing Machinery, pp. 549–562. ISSN: 0362-1340. DOI: 10.1145/1925844.1926449. URL: https://doi.org/10.1145/1925844.1926449.

[65] Joseph N Zadeh et al. "NUPACK: Analysis and design of nucleic acid systems". In: *Journal of computational chemistry* 32.1 (2011). Publisher: Wiley Subscription Services, Inc., A Wiley Company Hoboken, pp. 170–173.

[66] P. Du et al. "Algorithm-based fault tolerance for dense matrix factorizations". In: *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*. PPoPP '12. New York, NY, USA: ACM, 2012, pp. 225–234.

[67] A. Hwang, I. Stefanovici, and B. Schroeder. "Cosmic rays don't strike twice: understanding the nature of DRAM errors and the implications for system design". In: *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XVII. New York, NY, USA: ACM, 2012, pp. 111–122.

[68] H. Kaul et al. "Near-threshold voltage (NTV) design: Opportunities and challenges". In: *Proceedings of the 49th Annual Design Automation Conference*. DAC '12. New York, NY, USA: Association for Computing Machinery, 2012, pp. 1153–1158.

[69] Panruo Wu et al. "On-line soft error correction in matrix–matrix multiplication". In: *Journal of Computational Science*. Scalable Algorithms for Large-Scale Systems Workshop (ScalA2011), Supercomputing 2011 4.6 (Nov. 2013). Number: 6, pp. 465–472. ISSN: 1877-7503. DOI: 10.1016/j.jocs.2013.05.002. URL: https://www.sciencedirect.com/science/article/pii/S1877750313000641 (visited on 05/30/2024).

[70] Tomofumi Yuki et al. "AlphaZ: A System for Design Space Exploration in the Polyhedral Model". en. In: *Languages and Compilers for Parallel Computing*. Ed. by Hironori Kasahara and Keiji Kimura. Berlin, Heidelberg: Springer, 2013, pp. 17–31. ISBN: 978-3-642-37658-0. DOI: 10.1007/978-3-642-37658-0_2.

[71] Uday Bondhugula et al. "Tiling and optimizing time-iterated computations over periodic domains". In: *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*. 2014, pp. 39–50. DOI: 10.1145/2628071.2628106.

[72] C. Di Martino et al. "Lessons learned from the analysis of system failures at petascale: the case of blue waters". In: *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. 2014, pp. 610–621.

[73] S. Tavarageri, S. Krishnamoorthy, and P. Sadayappan. "Compiler-assisted detection of transient memory errors". In: PLDI '14. event-place: Edinburgh, United Kingdom. New York, NY, USA: Association for Computing Machinery, 2014, pp. 204–215.

[74] Doug Hakkarinen, Panruo Wu, and Zizhong Chen. "Fail-Stop Failure Algorithm-Based Fault Tolerance for Cholesky Decomposition". In: *IEEE Transactions on Parallel and Distributed Systems* 26.5 (May 2015). Number: 5 Conference Name: IEEE Transactions on Parallel and Distributed Systems, pp. 1323–1335. ISSN: 1558-2183. DOI: 10.1109/TPDS.2014.2320502. URL: https://ieeexplore.ieee.org/abstract/document/6805637 (visited on 05/30/2024).

[75] Dingwen Tao et al. "New-Sum: A Novel Online ABFT Scheme For General Iterative Methods". In: *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*. HPDC '16. event-place: Kyoto, Japan. New York, NY, USA: Association for Computing Machinery, 2016, pp. 43–55. ISBN: 978-1-4503-4314-5. DOI: 10.1145/2907294.2907306. URL: https://doi.org/10.1145/2907294.2907306.

[76] G. Aupy et al. "Coping with silent errors in HPC applications". In: *Emergent Computation : A Festschrift for Selim G. Akl*. Cham: Springer International Publishing, 2017, pp. 269–292. DOI: 10.1007/978-3-319-46376-6_11.

[77] Yufei Ding and Xipeng Shen. "GLORE: generalized loop redundancy elimination upon LER-notation". In: *Proceedings of the ACM on Programming Languages* 1.OOPSLA (Oct. 2017). Number: OOPSLA, 74:1–74:28. DOI: 10.1145/3133898. URL: https://dl.acm.org/doi/10.1145/3133898 (visited on 05/29/2024).

[78] M. Gupta et al. "Compiler techniques to reduce the synchronization overhead of GPU redundant multithreading". In: *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*. 2017, pp. 1–6. DOI: 10.1145/3061639.3062212.

[79] Sylvain Boulmé et al. "The Verified Polyhedron Library: an Overview". In: *2018 20th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*. Sept. 2018, pp. 9–17. DOI: 10.1109/SYNASC.2018.00014. URL: https://ieeexplore.ieee.org/document/8750763 (visited on 05/30/2024).

[80] Teresa Davies and Zizhong Chen. "Correcting soft errors online in LU factorization". In: *Proceedings of the 22nd International Symposium on High-Performance Parallel and Distributed Computing*. HPDC '13. event-place: New York, New York, USA. New York, NY, USA: Association for Computing Machinery, 2018, pp. 167–178. ISBN: 978-1-4503-1910-2. DOI: 10.1145/2462902.2462920. URL: https://doi.org/10.1145/2462902.2462920.

[81] O. Ovcharenko and V. Kazei. *Simple FDTD wave propagation in MATLAB*. 2018. URL: https://github.com/ovcharenkoo/WaveProp_in_MATLAB/tree/master/acoustic_2D_FDTD_wave_propagation.

[82] A. Cavelan and F. Ciorba. "Algorithm-Based Fault Tolerance for Parallel Stencil Computations". In: *2019 IEEE International Conference on Cluster Computing (CLUSTER)*. 2019.

[83] Mark E. Fornace, Nicholas J. Porubsky, and Niles A. Pierce. "A Unified Dynamic Programming Framework for the Analysis of Interacting Nucleic Acid Strands: Enhanced Models, Scalability, and Speed". In: *ACS Synthetic Biology* 9.10 (Oct. 2020). Number: 10 Publisher: American Chemical Society, pp. 2665–2678. DOI: 10.1021/acssynbio.9b00523. URL: https://doi.org/10.1021/acssynbio.9b00523 (visited on 05/20/2024).

[84] S. Höeffgen, S. Metzger, and M. Steffens. "Investigating the effects of cosmic rays on space electronics". In: *Frontiers in Physics* 8 (2020).

[85] L. Juracy et al. "A survey of aging monitors and reconfiguration techniques". In: *CoRR* abs/2007.07829 (2020).

[86] T. Marty, T. Yuki, and S. Derrien. "Safe overclocking for CNN accelerators through algorithm-level error detection". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39.12 (2020), pp. 4777–4790.

[87] H. D. Dixit et al. "Silent Data Corruptions at Scale". In: *CoRR* abs/2102.11245 (2021).

[88] OpenMP Architecture Review Board. "{OpenMP} Application Program Interface Version 5.2". In: Nov. 2021, pp. 124–140. URL: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf.

[89] Cambridge Yang, Eric Atkinson, and Michael Carbin. "Simplifying Dependent Reductions in the Polyhedral Model". In: *Proc. ACM Program. Lang.* 5.POPL (Jan. 2021). Place: New York, NY, USA Publisher: Association for Computing Machinery. DOI: 10.1145/3434301. URL: https://doi.org/10.1145/3434301.

[90] Kai Zhao et al. "FT-CNN: Algorithm-Based Fault Tolerance for Convolutional Neural Networks". In: *IEEE Transactions on Parallel and Distributed Systems* 32.7 (July 2021). Number: 7 Conference Name: IEEE Transactions on Parallel and Distributed Systems, pp. 1677–1689. ISSN: 1558-2183. DOI: 10.1109/TPDS.2020.3043449. URL: https://ieeexplore.ieee.org/abstract/document/9311863 (visited on 05/20/2024).

[91] Xavier Allamigeon, Ricardo D. Katz, and Pierre-Yves Strub. "Formalizing the Face Lattice of Polyhedra". In: *Logical Methods in Computer Science* Volume 18, Issue 2 (May 2022). Publisher: Episciences.org. ISSN: 1860-5974. DOI: 10.46298/lmcs-18(2:10)2022. URL: https://lmcs.episciences.org/9570 (visited on 05/30/2024).

[92] Louis Narmour, Steven Derrien, and Sanjay Rajopadhye. "Automatic Algorithm-Based Fault Tolerance (AABFT) of Stencil Computations". In: *2023 32nd International Conference on Parallel Architectures and Compilation Techniques (PACT)*. Oct. 2023, pp. 187–198. DOI: 10.1109/PACT58117.2023.00024. URL: https://ieeexplore.ieee.org/abstract/document/10364587 (visited on 05/20/2024).

[93] N. Zechar. *FDTD 3D and 2D Examples - ADE CPML and a Dielectric Region*. en. Oct. 2024. URL: https://www.mathworks.com/matlabcentral/fileexchange/101664-fdtd-3d-and-2d-examples-ade-cpml-and-a-dielectric-region (visited on 10/03/2024).