Fourier-Motzkin with Non-Linear Symbolic Constant Coefficients

by

Patricia A. Suriana

S.B., Massachusetts Institute of Technology (2014)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2016

Author .....
Department of Electrical Engineering and Computer Science
January 25, 2016

Certified by .....
Saman P. Amarasinghe
Professor
Thesis Supervisor

Accepted by .....
Christopher J. Terman
Chairman, Master of Engineering Thesis Committee

Fourier-Motzkin with Non-Linear Symbolic Constant Coefficients

by

Patricia A. Suriana

Submitted to the Department of Electrical Engineering and Computer Science
on January 25, 2016, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

The polyhedral framework is an elegant and useful system for reasoning about loop nests in programs, and is commonly used to perform complex loop transformations such as tiling and parallelization. However, several critical transformations introduce non-linear inequalities during code generation, which present difficulties for the polyhedral model. Proposals for extending the framework to deal with non-linear inequalities have generally been complex and are not used in current code generators. We propose a simple extension to Fourier-Motzkin elimination that deals with the specific case of non-linearity arising from symbolic constant coefficients, and show that this enables the polyhedral framework to deal with important cases that commonly occur in code generation. We build a framework, called NFM, that implements the extension and integrate the new system into Halide, an open-source domain-specific language compiler for image processing [13], which provides a more robust framework to perform computation on iteration domain such as merge, intersection, etc., and provides Halide a unified framework to perform more complex optimization schemes, such as diamond tiling.

Thesis Supervisor: Saman P. Amarasinghe
Title: Professor

Acknowledgments

At the end of my thesis, I would like to thank all those people who made this thesis possible and an unforgettable experience for me.

I would first like to thank my thesis advisor Saman Amarasinghe for his excellent guidance, caring, patience, and providing me with an excellent atmosphere for doing research.

I would also like to acknowledge Riyadh Baghdadi as the second reader of this thesis, and I am gratefully indebted to him for his very valuable comments on this thesis. I am also thankful for his technical assistance to my project.

I am thankful to the other members of the COMMIT group, particularly Shoaib Kamil for his feedback and assistance during the early development of NFM, and Tyler Denniston for his help on distributed Halide.

Finally, I must express my very profound gratitude to my parents and to my two brothers for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them. Thank you.

THIS PAGE INTENTIONALLY LEFT BLANK

Contents

List of Figures

List of Tables

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 1

Introduction

Optimizing compilers for high performance computing generally perform complex loop transformations, such as tiling, a compiler optimization for improving cache utilization, and parallelization. Thus, generating loop nests after complex loop transformations is an important component of these compilers. While there are many methods for final code generation, one of the most powerful and promising techniques is using a linear polyhedral framework.

In polyhedral framework, each loop iteration within nested loops corresponds to a lattice point inside an integer polyhedron, or better known as $\mathbb{Z}$-polyhedron. The bound of this polyhedron is defined by a set of inequality/equality constraints. Loop transformations, such as loop reordering, tiling, etc., are equivalent to affine transformations on the polyhedron.

Although the polyhedral framework handles many complex array indices and loop bounds, one of the critical requirements is that all access functions must be affine. However, several critical transformations, such as parametric tiling, where the tile size is not known at compile time by remaining a symbolic parameter, and parallelization for distributed memory machines, involve solving non-linear (hence non-affine) inequalities during code generation.

Consider the two-nested loops shown in Figure 1-1 as an example. In the polyhedral framework, the iteration domain of the system is represented as $S = \{(i, j) : 0 \leq i \leq U \ \wedge \ 0 \leq j \leq min(2i, V)\}$. To speed up the execution time, we distribute

```
for i = 0 to U
  for j = 0 to min(2i, V)
```

Figure 1-1: Example loop nest Program 1

iterations of the inner loop j in equal-sized blocks across multiple processors to be run in parallel (see Figure 1-2). To describe this new iteration space, we introduce the following constraints: $bx \leq j \leq bx + b - 1$, where b is the block size and x is the processor ID. If b is unknown at compile time, we end up with non-linear system.

This class of problems, better known as parameterized tiled code generation, is important. The performance gain due to loop tiling is directly related to the tile size. Often, choosing a tile size that results in good performance is difficult. In that case, an autotuning system is used to find the optimum tile size. Autotuning automatically and empirically searches a space of parameter values and selects the one that provides the best execution time. With fixed tile sizes, we will need to regenerate and recompile the code for each tile size, but if the tile size is left as a symbolic constant, we only need to generate and compile the code once for any tile size. For parallelized code, the number of processors is usually unknown until run-time; therefore, it is necessary to leave the processor number/ID as a symbolic constant during code generation.

1.1 Background and Related Work

The polyhedral framework [7, 9, 10] refers to a general model for optimizing programs with loops and arrays, in which loop iterations are considered to be points in a multidimensional geometric space and loop optimizations exploit properties of the polyhedra to transform the loop while preserving correctness. Optimization techniques using the polyhedral model are widely used in compilers and continue to be an area of active research.

Several attempts to deal directly with non-linear constraints during code generation have been made in the past. Amarasinghe [2] extended Fourier-Motzkin elimination to handle a restricted form of symbolic constant coefficients in the inequalities,

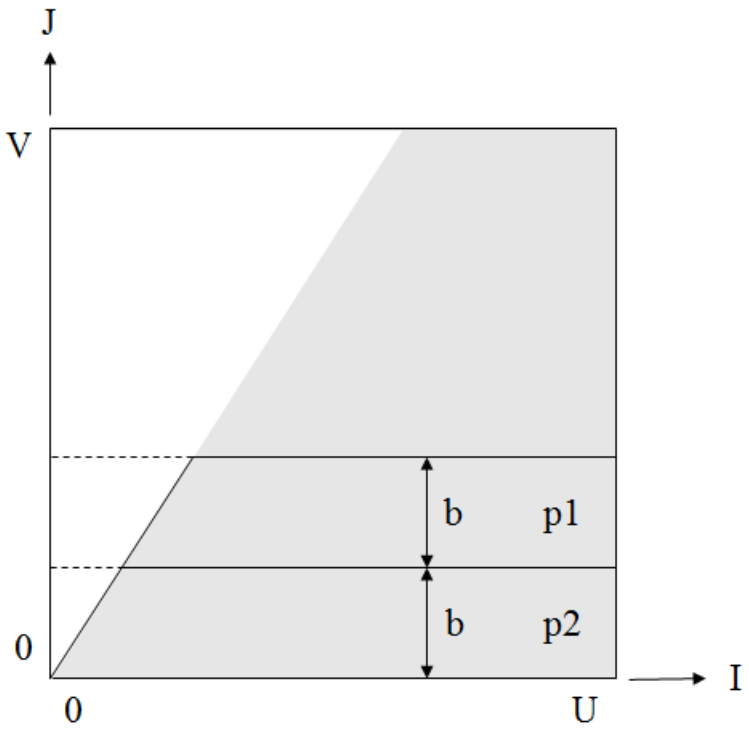

Figure 1-2: Rectangular tiling on loop j of Program 1

in which the symbolic constant coefficients have to be in linear polynomial form; however, full details on the approach were not made available. Größlinger et al. [8] proposed an extension to the polyhedral model, in which they allow arbitrary rational polynomials as coefficients in the inequalities that define the iteration space of the loop. However, their method uses quantifier elimination over real algebra to reduce the number of branches that arise during loop generation, which is computationally expensive.

Despite previous work, the current state of the art in code generation does not seem to incorporate these non-linear extensions. Some current approaches, such as [14] and [15], deal with parameterized tile sizes by splitting the problem into subproblems of generating loops that iterate over tiles and then loops that iterate over points within tiles. These techniques are at best an ad-hoc extension to the elegant polyhedral framework. In addition, they only deal specifically with rectangular tiling. Once the tile shape changes, e.g. diamond or hexagonal tiling, the methods no longer work.

In this paper, we revisit a non-linear extension to Fourier-Motzkin elimination (FME) with symbolic constant coefficients. Unlike in [2], the coefficients can be of any integer polynomial degree. We limit our scope to integer polynomial symbolic coefficients, unlike in [8], which allows more general rational polynomials. Since our main use case of FME is for loop generation, the integer assumption makes sense; all loop bounds and array accesses are integers and we can convert rational coefficients into integers by multiplying by their denominators. This restriction simplifies the implementation and allows us to perform additional steps to simplify the constraints without sacrificing the system's capability for loop generation.

In Chapter 4, we show that by adding this small extension to Fourier-Motzkin elimination to deal with non-linear symbolic constant coefficients, we are able to handle many examples that leave tile/block sizes unknown at compile time within the same framework, described in the next chapter.

1.2 Contributions

Some of the widely used polyhedral model based compilers, such as Pluto [5] which uses ClooG [4] as back-end for code generation, only handle code with affine transformations. Pluto can generate tiled code, but only if the tile size is fixed at compile time. In this paper, we revisit a non-linear extension to Fourier-Motzkin elimination for the polyhedral model with symbolic constant coefficients. We will show that by adding a small extension to Fourier-Motzkin elimination to deal with non-linear symbolic constant coefficients, we are able to handle various cases, such as rectangular tiling, multi-level tiling, communication code generation, and diamond tiling — all of which leave tile/block sizes unknown at compile time — using the same framework.

We build a framework, called NFM, that implements the extension and build a simple loop code generator on top of the framework. Further, we integrate the NFM framework into Halide, an open-source domain-specific language compiler for image processing [13], which provides Halide a more robust framework to perform computation on iteration domain such as merge, intersection, etc., and a unified

framework to perform more complex scheduling schemes, such as diamond tiling.

1.3 Organization

The paper is structured as follows. Chapter 1.1 introduces the polyhedral model and discusses related work. Chapter 2 describes the extended Fourier-Motzkin elimination method with non-linear symbolic coefficients, which we will refer to as NFM. Chapter 3 describes some processing steps to eliminate redundant inequalities generated by NFM. In Chapter 4, we provide results of running NFM on various examples, including multi-level rectangular tiling, communication code generation, and diamond tiling. In Chapter 5, we discuss the integration of NFM into Halide and some performance gain of using NFM over Halide's `simplify` method to simplify loop bound. Finally, we summarize and conclude with Chapter 6.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 2

Extended Fourier-Motzkin Elimination

During code generation, we compute the bounds of a loop nest by scanning the polyhedron describing the iteration domain of the loop nest in reverse order from innermost to outermost loop. The bound of a loop nest is such that it only depends on the enclosing loop variables, symbolic constants, and integer constants. Inner loop bound may depend on outer loop variables, however, outer loop bound should not depend on inner loop variables. To determine loop bounds for outer loops, we project out inner loop iteration variables via Fourier-Motzkin elimination method.

2.1 Fourier-Motzkin Elimination Method

We briefly review the Fourier-Motzkin elimination (FME) method [6]. Consider the linear system $Ax + b \geq 0$ where $A \in \mathbb{Z}^{m,n}$ and $b \in \mathbb{Z}^m$:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n + b_1 \geq 0 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n + b_2 \geq 0 \\ \cdots\cdots\cdots\cdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n + b_m \geq 0 \end{aligned} \tag{2.1}$$

where m is the total number of inequalities and n is the total number of integer variables in the system. Suppose that we are eliminating x_1 from the system. We can generate an equivalent system by multiplying by $1/|a_{i1}|$ for each inequality i where a_{i1} is non-zero and grouped them into three classes depending on the sign (positive, negative or zero) of the coefficient for x_1:

$$\begin{aligned} x_1 + a'_{i2}x_2 + \cdots + a'_{in}x_n + b'_1 \geq 0 \quad & (i \in S^+) \\ a_{i2}x_2 + \cdots + a_{in}x_n + b_i \geq 0 \quad & (i \in S^0) \\ -x_1 + a'_{i2}x_2 + \cdots + a'_{in}x_n + b'_i \geq 0 \quad & (i \in S^-) \end{aligned} \tag{2.2}$$

where $S^+ = \{i \mid a_{i1} > 0\}$, $S^0 = \{i \mid a_{i1} = 0\}$, $S^- = \{i \mid a_{i1} < 0\}$, $a'ij = a_{ij}/|a_{i1}|$, and $b'i = b_i/|a_{i1}|$. $x_1, x_2, x_3, \cdots, x_n$ is a solution to the original system 2.1 if and only if $x_2, x_3, \cdots, x_n$ satisfies:

$$\begin{aligned} \sum_{j=2}^{n} a'_{kj}x_j + b'_k + \sum_{j=2}^{n} a'_{ij}x_j + b'_i \geq 0 \quad & (i \in S^+, k \in S^-) \\ \sum_{j=2}^{n} a'_{ij}x_j + b'_k \geq 0 \quad & (i \in S^0) \end{aligned} \tag{2.3}$$

and x_1 satisfies:

$$\max_{i \in S^+} \left(\sum_{j=2}^{n} -a'_{ij}x_j - b'_i \right) \leq x_1 \leq \min_{k \in S^-} \left(\sum_{j=2}^{n} a'_{kj}x_j + b'_k \right) \tag{2.4}$$

The polyhedron defined by 2.3 is the projection of the original polyhedron 2.1 onto the space of $x_2, x_3, \cdots, x_n$, whereas, 2.4 defines the lower and upper bound of x_1 in terms of $x_2, x_3, \cdots, x_n$. We proceed similarly to eliminate $x_2, x_3, \cdots, x_n$.

2.2 Non-Linear Symbolic Constant Coefficient Extension

We extend the Fourier-Motzkin elimination (FME) method to handle non-linear systems with symbolic constant coefficients. We will refer to the extended system as

NFM throughout the rest of the paper. The coefficients of the variables of the linear inequalities can be an integer polynomial in the symbolic constants. More formally, the inequality is of the form:

$$f(\vec{u}, \vec{v}) = p_0(\vec{u}) + p_1(\vec{u})v_1 + p_2(\vec{u})v_2 + \cdots + p_n(\vec{u})v_n \geq 0$$

where $\vec{v} = (v_1, v_2, \ldots, v_n)$ are integer variables and $p_0(\vec{u})$, $p_1(\vec{u})$, $\ldots$, $p_n(\vec{u})$ are arbitrary integer polynomials in the symbolic constants $\vec{u} = (u_1, u_2, \ldots, u_m)$. Specifically,

$$\forall x(0 \leq x \leq n)\ p_x(\vec{u}) = \sum_{i=0} c_{x,i} \prod_{j=1}^{m} u_{x,j}^{z_{x,ij}},$$

where $c_{x,i}$ is integer constant and $z_{x,ij}$ is the polynomial power of the symbolic constant $u_{x,j}$.

Consider the following system of inequalities:

$$\begin{aligned}
f^1(\vec{u}, \vec{v}) &= p_0^1(\vec{u}) + p_1^1(\vec{u})v_1 + p_2^1(\vec{u})v_2 + \cdots + p_n^1(\vec{u})v_n \geq 0 \\
f^2(\vec{u}, \vec{v}) &= p_0^2(\vec{u}) + p_1^2(\vec{u})v_1 + p_2^2(\vec{u})v_2 + \cdots + p_n^2(\vec{u})v_n \geq 0 \\
&\cdots\cdots \\
f^r(\vec{u}, \vec{v}) &= p_0^r(\vec{u}) + p_1^r(\vec{u})v_1 + p_2^r(\vec{u})v_2 + \cdots + p_n^r(\vec{u})v_n \geq 0
\end{aligned}$$

where r is the total number of inequalities in the system. Without loss of generality, suppose that we are eliminating the variable v_x. First, we separate the inequality constraints into four sets: S_x^0, S_x^+, S_x^-, and $S_x^?$. S_x^0 contains any inequalities not involving v_x, i.e.

$$S_x^0 = \{f^y(\vec{u}, \vec{v}) \ \mid\ 0 \leq y \leq r,\ p_x^y = 0\}.$$

S_x^+ contains any inequalities which coefficient values of v_x are positive, i.e.

$$S_x^+ = \{f^y(\vec{u}, \vec{v}) \ \mid\ 0 \leq y \leq r,\ p_x^y > 0\}.$$

S_x^- contains any inequalities which coefficient values of v_x are negative, i.e.

$$S_x^- = \{f^y(\vec{u}, \vec{v}) \mid 0 \leq y \leq r,\ p_x^y < 0\}.$$

Finally, $S_x^?$ contains any inequalities in which coefficient values of v_x are undetermined.

In addition to the four sets of inequality constraints, we also have a context, D, which contains partial domain knowledge of the symbolic constant values, e.g. $-10+u_1 \geq 0$ or $u_1 * u_2 + 10u_1 \geq 0$. More formally, the context consists of inequalities (either linear or non-linear) of the form

$$g(\vec{u}) = \sum_{i=0} c_i \prod_{j=1}^{m} u_j^{z_{ij}},$$

where $\vec{u} = (u_1, u_2, \ldots, u_m)$ are symbolic constants, c_i is integer constant and z_j is the polynomial power of the symbolic constant u_j.

Given S_x^0, S_x^+, S_x^-, $S_x^?$, and D, we proceed as follows. In linear FM, to eliminate v_x from the inequality constraints, we take pair of constraints with opposite coefficient values of v_x and compute their positive combination (with appropriate positive constant multipliers) to create a new inequality constraint not involving v_x. Since the value of p_x^y $(0 \leq y \leq r \ \wedge \ f^y(\vec{u}, \vec{v}) \in S_x^?)$ is undetermined, for every inequality in $S_x^?$, we recursively branch on the current system into the following cases: (1) $p_x^y = 0$, (2) $p_x^y > 0$, and (3) $p_x^y < 0$. If $p_x^y = 0$, we move the inequality from $S_x^?$ to S_x^0. Similarly if $p_x^y > 0$ or $p_x^y < 0$, we move the inequality from $S_x^?$ to S_x^+ or S_x^- respectively.

Each branch introduces a new set of linear constraints on the symbolic parameters to the context. For example, let's suppose $p_x^1 = 8 - 4N$, where N is a symbolic parameter. For the case when $p_x^1 > 0$, constraint $N < 2$ is added to the context D. Similarly, for the case when $p_x^1 < 0$, constraint $N > 2$ is added to the context. If adding new constraints results in an empty context domain, the system is not feasible and we can drop the corresponding branch. Since fully checking the feasibility of D involves more non-linear inequalities, we approximate the checking by employing regular Fourier-Motzkin elimination method on only the linear inequalities in D. If

the sub-domain defined by the linear inequalities is empty, we can conclude that D is empty as well.

After every element in $S_x^?$ is classified into either S_x^0, S_x^+, or S_x^-, we take one element $f^+(\vec{u},\vec{v})$ from S_x^+ and one element $f^-(\vec{u},\vec{v})$ from S_x^- and compute their greatest common divisor, $gcd = GCD(p_x^+, -p_x^-)$. Note that in linear system, gcd is a constant. Multiplying $f^+(\vec{u},\vec{v})$ by $-p_x^-/gcd$ and $f^-(\vec{u},\vec{v})$ by p_x^+/gcd, and adding those together produces a new inequality not involving v_x. We repeat the same steps for every combination of each element in S_x^+ and S_x^- to eliminate v_x from all the inequality constraints.

To illustrate the steps described above, let's eliminate j from the system of inequalities describing the iteration domain of Program 1. The system of inequalities describing the iteration space is listed in Figure 2-1. Let's also assume we have an additional context D on the symbolic constants, $D = \{(U,V) \mid U \geq 0,\ V \geq 0,\ b \geq 1\}$, which is a reasonable assumption since i and j start from 0 and it doesn't make sense to have a non-positive block size. First, we separate the inequalities into $S_j^0 = \{(1),(2)\}$, $S_j^+ = \{(3),(6)\}$, $S_j^- = \{(4),(5),(7)\}$, and $S_j^? = \emptyset$. $S_j^?$ is empty since all coefficients of j have signs that are fully determined. Next, we compute the positive combinations of elements in S_j^+ and S_j^- which create new inequalities that do not involve j. Take as an example inequalities (3) and (4). We multiply both (3) and (4) by 1 and sum them together, creating a new inequality $2 * i \geq 0$.

The new set of inequalities after projecting out j (i.e. eliminating j from all the constraints) is listed in Figure 2-2. Inequalities $V \geq 0$ and $b - 1 \geq 0$ only involve the symbolic constants, b and V. Since we know from the context D that $V \geq 0$ and $b \geq 1$, we can safely drop those inequalities from the system; otherwise, we need to add the constraints $V \geq 0$ and $b - 1 \geq 0$ to D. Had the context specified $b < 1$ or $V < 0$, the system would be infeasible and we could return immediately. Inequality (10) is redundant in the presence of (8) and we can thus remove it from the system. Chapter 3 describes several steps to simplify the system of inequalities. Next, we proceed to project out i and x in that order to determine the loop bounds of i and x in terms of the outer loop integer variables and symbolic constants. The generated

$$i \geq 0 \tag{2.5}$$
$$-i + U \geq 0 \tag{2.6}$$
$$j \geq 0 \tag{2.7}$$
$$2 * i - j \geq 0 \tag{2.8}$$
$$-j + V \geq 0 \tag{2.9}$$
$$-b * x + j >= 0 \tag{2.10}$$
$$b * x - j + b - 1 \geq 0 \tag{2.11}$$

Figure 2-1: System of inequalities describing the iteration space of Program 1 with parallelization and blocking

$$i \geq 0 \tag{2.12}$$
$$-i + U \geq 0 \tag{2.13}$$
$$2 * i \geq 0 \tag{2.14}$$
$$V \geq 0 \tag{2.15}$$
$$b * x + b - 1 \geq 0 \tag{2.16}$$
$$-b * x + 2 * i \geq 0 \tag{2.17}$$
$$-b * x + V \geq 0 \tag{2.18}$$
$$b - 1 \geq 0 \tag{2.19}$$

Figure 2-2: System of inequalities after projecting out dimension j from the domain set in Figure 2-1

loop code is shown in Figure 2-3.

For now, our code generator only covers cases where everything divides and is a single statement. We believe that it will be possible to handle other cases as well (e.g. multiple statements, non-divisible block size, etc.), however, we have not yet implemented those functionalities into our code generator.

```
IF ((U >= 0) AND (V >= 0) AND (b-1 >= 0)) THEN
  FOR x := max(0, ((-b+1)/b)) TO
           min(2*U/b, V/b) DO
    FOR i := max(b*x/2, 0) TO U DO
      FOR j := max(0, b*x) TO min(2*i, V, b*x+b-1) DO
```

Figure 2-3: Rectangular tiling on Program 1

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 3

Simplifying the Constraints

On each step, Fourier-Motzkin elimination produces a large number of redundant constraints. To prevent explosion in the number of constraints and to make the system more manageable to solve, we iterate over all the inequalities created during each elimination step, removing as many redundant constraints as possible.

3.1 Gaussian Elimination

We scan over the inequalities; if we find pair of inequalities with opposite signs (e.g. $Bx + 1 \geq 0$ and $-Bx - 1 \geq 0$), we convert them into an explicit equality. We then perform gaussian elimination on the equalities and back substitution of the solution to remove some of the integer variables $(v_1, v_2, \ldots, v_n)$ from the inequalities. For example, consider the following system of inequalities:

$$x - B \geq 0$$

$$-x + B \geq 0$$

$$x + y - 2 \geq 0$$

where B is a symbolic constant. $x - B \geq 0$ and $-x + B \geq 0$ are pairs of implicit equality; they are converted into $x - B = 0$. Back substituting $x - B = 0$ into $x + y - 2 \geq 0$ gives $y + (B - 2) \geq 0$ as the new inequality. The simplified domain is then,

$$x - B = 0$$

$$y + B - 2 \geq 0$$

3.2 Normalizing the Constraints

Next, we simplify the equalities/inequalities by dividing all the symbolic coefficients (including the constant terms) by their greatest common integer divisor, GCD_{all}. Next, since we are only interested in an integer solution for code generation, we can simplify the inequalities further by dividing all the integer variables' coefficients by their greatest common integer divisor, GCD_{iter}, and rounding off the constant term offset to the nearest value if the constant term offset is an integer (not a symbolic constant). If the constant term offset is a symbolic constant, it may or may not be divisible by GCD_{iter}. Had it been divisible by GCD_{iter}, rounding off the constant term offset will produce incorrect result.

The simplification of an inequality $2Nx + 4Ny + 9 \geq 0$, where N is a symbolic constant, involves the following steps:

1. Compute the integer GCD of all coefficients: $GCD_{all} = gcd(2N, 4N, 9) = 1$.

2. Divide each coefficient by GCD_{all}, which is equal to one in this case.

3. Since it is an inequality, we could further divide the constraint by the integer GCD of the non-constant terms's coefficients: $GCD_{iter} = gcd(2N, 4N) = 2$. Note that even though N is also a common factor for both $2Nx$ and $4Ny$, we do not divide the constraint by $2N$ as N may be positive or negative or zero. If N is negative, dividing by N will reverse the inequality. Likewise, if N is zero, the division is undefined.

4. The final result is $Nx + 2Ny + \lceil 9/2 \rceil \geq 0$ or $Nx + 2Ny + 5 \geq 0$.

For equality constraint where all of its coefficients are integer, if the constant term is not divisible by GCD_{iter}, we can conclude that the system is infeasible since we operate in the integer domain.

3.3 Removing Redundant Constraints

We use a simple algorithm described in [2] with some modifications to remove some of the redundant inequalities. Given a system of inequalities S, for all pairs of inequalities

$$\{p_0^1(\vec{u}) + p_1^1(\vec{u})v_1 + \cdots + p_n^1(\vec{u})v_n \geq 0\} \in S$$

and

$$\{p_0^2(\vec{u}) + p_1^2(\vec{u})v_1 + \cdots + p_n^2(\vec{u})v_n \geq 0\} \in S$$

where $\forall i(1 \leq i \leq n)\ p_i^1(\vec{u}) == p_i^2(\vec{u})$, we remove $p_0^1(\vec{u}) + p_1^1(\vec{u})v_1 + \cdots + p_n^1(\vec{u})$ from S if $p_0^1(\vec{u})$ can be determined to be equal to or greater than $p_0^2(\vec{u})$. Otherwise, if $p_0^2(\vec{u})$ can be determined to be greater than $p_0^1(\vec{u})$, we remove $p_0^2(\vec{u}) + p_1^2(\vec{u})v_1 + \cdots + p_n^2(\vec{u})v_n \geq 0$ from S.

In addition to the method outlined above, we perform an additional redundancy check by replacing the constraint in question with its negation. If the new system does not have an integer solution, then the constraint is redundant and thus can be eliminated. Since this check involves solving a new system of inequalities using NFM, we only perform the check at the very last step of simplification.

The three steps listed above help eliminate a number of equalities/inequalities in the system on each FME step, which in turn reduces the memory and time complexity of the solver. In addition, for programs with complicated loop bounds, the order of elimination may have impact on performance; some elimination orders may produce much simpler final loop bounds — bounds with fewer floor/ceiling operations and fewer inequalities involved in the lower/upper bounds of a loop iteration. Given the constraints on the symbolic constants and the loop bounds of the outer loop iterations, it is also possible to further simplify the loop bounds of one loop iteration. Consider Program 2 in Figure 3-1. The value of p_r is between 1 and $N/32$ inclusive, and its step size is one, thus, the value of $32 * p_r$ is bounded below by 1 and above by N. Since we know that $N \geq 32$, we could simplify the lower bound of i_r into $32 * p_r$. We

```
IF (N-32 >= 0) THEN
  FOR pr := 1 TO (N/32) DO
    FOR ir := max(32*pr, 6) TO min(32*pr+2, N) DO
```

Figure 3-1: Example loop nest Program 2

plan on incorporating these additional simplification steps into our code generator.

Chapter 4

Examples

Extending Fourier-Motzkin elimination (FME) to handle symbolic constant coefficients allows us to solve many other problems not limited to simple blocking. To demonstrate the applicability of our system, we apply the extended FME system on more complex examples, including multi-level rectangular tiling, communication code generation, and diamond tiling. We believe that these examples demonstrate that the extension to non-linear systems with symbolic constant coefficients can handle modern complexities of code generation without compromising the ability to have symbolic bounds and numbers of processors.

4.1 Multi-Level Rectangular Tiling

Using the previous Program 1 in Chapter 1 as an example, we would like to perform two-level tiling on dimension i (see Figure 4-1 for the iteration space diagram). The iteration space set is listed in Figure 4-2. B_1 and B_2 are the tile sizes for level 1 and level 2 tiling respectively, while t_1 and t_2 are the intra-tile iterators for level 1 and level 2 tiling. The generated loop code after projecting out the loop iterators in reverse order from inner to outer loop nests (j, i, t_2, and t_1 in that order) is shown in Figure 4-3.

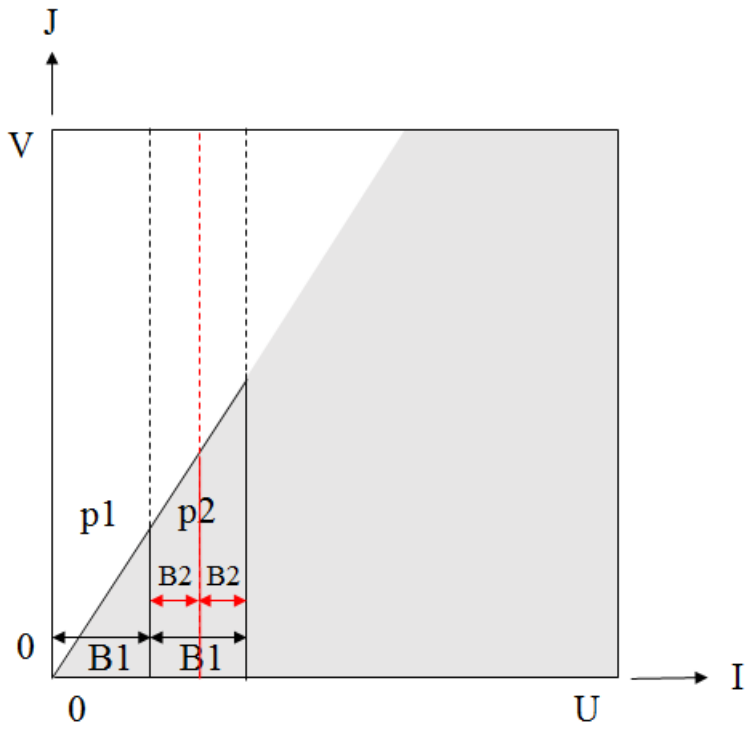

Figure 4-1: Two-level tiling on dimension i of Program 1

$$i \geq 0 \tag{4.1}$$
$$-i + U \geq 0 \tag{4.2}$$
$$j \geq 0 \tag{4.3}$$
$$2 * i - j \geq 0 \tag{4.4}$$
$$-j + V \geq 0 \tag{4.5}$$
$$-B_1 * t_1 + t_2 \geq 0 \tag{4.6}$$
$$B_1 * t_1 - t_2 + B_1 - 1 \geq 0 \tag{4.7}$$
$$-B_2 * t_2 + i \geq 0 \tag{4.8}$$
$$B_2 * t_2 - i + B_2 - 1 \geq 0 \tag{4.9}$$

Figure 4-2: System of inequalities describing the iteration space of Program 1 with two-level tiling on dimension i

```
IF ((U >= 0) AND (V >= 0) AND (B1-1 >= 0) AND (B2-1 >= 0) AND
   (B1*B2-B2+U >= 0)) THEN
  FOR t1 := max(0, (-B1+1)/B1, (-B1*B2+1)/(B1*B2)) TO
     (U/(B1*B2)) DO
    FOR t2 := max(0, (-B2+1)/B2, B1*t1) TO min(U/B2,
       B1*t1+B1-1)) DO
      FOR i := max(0, B2*t2) TO min(U, B2*t2+B2-1) DO
        FOR j := 0 TO min(2*i, V) DO
```

Figure 4-3: Two-Level Rectangular Tiling on Program 1

```
for t = 0 to T
  for i = 3 to N
    X[i] = X[i-3]
```

Figure 4-4: Communication Code for Program 3

4.2 Communication Code

We use a program from [1] as our next example (see Figure 4-4). As in [1], we would like to distribute the second loop in the program as blocks of B iterations across a linear array of processors and generate the necessary communication code between processors. However, instead of specifying a fixed block size at compile time, we keep B as a symbolic constant. Because of the data dependence relation, a processor p_s writing to $X[i_s]$ at time t_s needs to send the data to a processor p_r writing to $X[i_r]$ at time t_r given that $t_r = t_s$ and $i_r = i_s + 3$. Table 4.1 shows the set of equality/inequality constraints for the communication sets of the receiver and sender. The constraint $p_s \neq p_r$ in the communication set involves a disjunction. To handle this, we perform the computation twice: one with inequality $p_s > p_r$ and one with inequality $p_s < p_r$. We project the system in reverse order from inner to outer loop nests $(a, i_s, t_s, p_s, i_r, t_r, p_r)$ to generate the receiver loop (shown in Figure 4-5).

4.3 Diamond Tiling

We also tested our system on diamond tiling. We use the example appearing in Section IV.B of [3]. We set the block size as a symbolic constant B. The original program and

Context	$t_r \geq 0$	$T - t_r >= 0$
	$i_r - 3 \geq 0$	$N - i_r >= 0$
	$i_r - 6 \geq 0$	
	$t_s - t_r \geq 0$	$t_r - t_s >= 0$
	$i_s - i_r + 3 \geq 0$	$i_r - i_s - 3 >= 0$
Access Function	$i_r - 3 - a \geq 0$	$a - i_r + 3 >= 0$
Computation decomposition for read iterations	$i_r - Bp_r \geq 0$	$Bp_r + B - 1 - i_r >= 0$
Computation decomposition for write iterations	$i_s - Bp_s \geq 0$	$Bp_s + B - 1 - i_s >= 0$
Constraint $p_s \neq p_r$	$p_s > 0$	or $p_s < p_r$

Table 4.1: Inequalities defining the communication sets for receiver code in Program 3

```
IF ((N-6 >= 0) AND (B-1 >= 0) AND (T-1 >= 0) AND (B+N-7 >= 0))
   THEN
  FOR pr:= max(4/B, ((-B+7)/B)) TO N/B DO
    FOR tr := 0 TO T DO
      FOR ir := max(6, B*pr) TO min(B*pr+2, N, B*pr+B-1) DO
        FOR ps := ((ir-B-2)/B) TO min(pr-1, (ir-3)/B) DO
          ts := tr
          FOR is := max(ir-3, B*ps) TO min(ir-3, B*ps+B-1) DO
            a := ir-3
```

Figure 4-5: Communication Code for Program 3

```
for t = 0 to T
  for i = 1 to N
    for j = 1 to N
```

Figure 4-6: Example loop nest Program 4

$$
\begin{aligned}
T''((t,i,j)) = (t_1 + t_2, t_2, t_3, t, t+i, t+j \mid \\
B * t_1 \leq t + i \leq B * t_2 + B - 1, \\
B * t_1 \leq t - i \leq B * t_2 + B - 1, \\
B * t_3 \leq t + j \leq B * t_3 + B - 1)
\end{aligned}
$$

Figure 4-7: Diamond tiling transformation applied to Program 3

its diamond tiling transformation are shown in Figure 4-6 and Figure 4-7 respectively. To generate the appropriate loop code, we embed the equalities/inequalities describing the scheduling constraints into the system of constraints describing the iteration space of the system (see Section 3.1 of [4] for more detail). The generated code is shown in Figure 4-8.

```
IF ((T-1 >= 0) AND (N-2 >= 0) AND (B-2 >= 0)) THEN
  FOR tk := ((-2*B+2)/B) TO (2*T/B) DO
    FOR t2 := max(tk-(N+T)/B, tk/2-(B+2*N-1)/(2*B),
       (-B-N+1)/B) TO min(tk/2+(B-3)/(2*B), (T-1)/B) DO
      FOR t3 := max(tk-t2+(2-B-N)/B, tk/2+(2-B)/B, (2-B)/B,
         t2+(3-B)/B) TO min(tk-t2+(2-B+N)/B, tk/2+(B+N-1)/B,
         t2+(B+2*N-1)/B, (N+T)/B) DO
        FOR tp := max(0, B*t2+1, B*t3-N, B*tk-B*t2-N, B*tk/2)
           TO min(T, B*tk/2+B-1, B*tk-B*t2+B-2, B*t3+B-2,
           B*t2+B+N-1) DO
          FOR ip := max(B*tk-B*t2, -B*t2+2*tp-B+1, tp+1) TO
             min(B*tk-B*t2+B-1, -B*t2+2*t_p, tp+N) DO
            FOR jp := max(B*t3, tp+1) TO min(B*t3+B-1, tp+N) DO
```

Figure 4-8: Diamond Tiling on Program 4

Chapter 5

NFM Integration into Halide

We integrated our NFM framework into Halide, an open-source domain-specific language for image processing and computational photography [13]. In the integrated system, we use NFM to simplify interval expression after box merging or to compute box intersection, a critical path during loop bound computation, which we will describe in detail in the following sections.

5.1 Halide

Halide is a domain specific language designed specifically for image processing and computational photography. It is a functional programming language that allows programmer to specify image processing algorithm, without having to worry about how the algorithm is actually implemented. Halide decouples the algorithm from its schedule, which allows programmers to explore different scheduling strategies without having to modify the algorithmic code [12].

In Halide, a region is represented as a multidimensional interval (axis-aligned bounding box). The iteration domain of a loop nest is represented as simple axis-aligned bounding boxes of the dimensionality of the loop depth; for each loop level, its lower and upper bound are modeled as intervals [11]. Loop bound computation during loop synthesis is based on interval analysis. Interval analysis is more straightforward to implement compared to polyhedral model. It can analyze a relatively broad class

of expressions and can handle more generalized constructs such as parametric tiling, which are beyond the scope of linear polyhedral model.

5.2 Integrating NFM into Halide

Despite its capability to handle more general constructs with symbolic constants, such as seen in parametric tiling, the resulting lower/upper bounds for each loop dimension generated by interval analysis are generally less expressive compared to those generated by polyhedral model. Halide's iteration domain is constrained to axis-aligned bounding boxes, rather than arbitrary polytopes as in the polyhedral model.

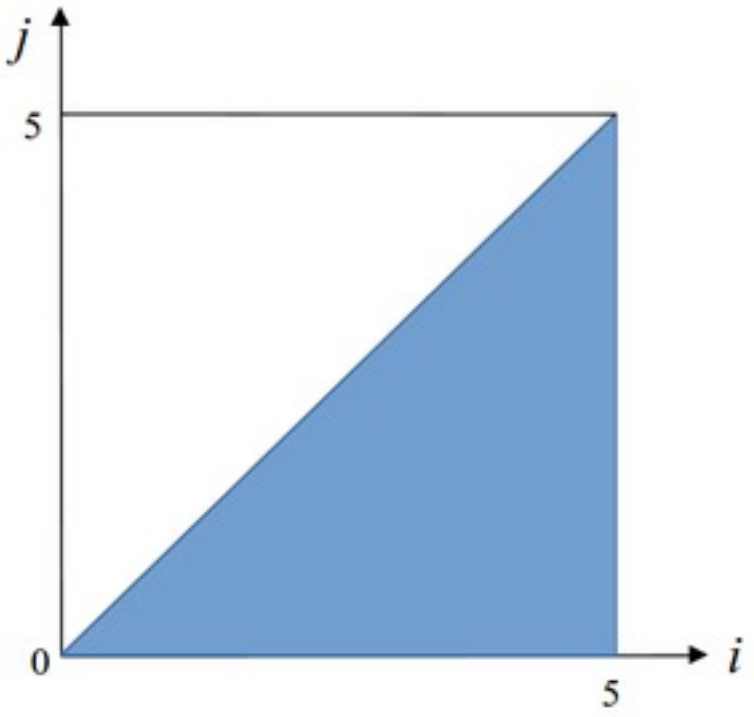

Figure 5-1: Triangular iteration domain of loop dimension i and j

To illustrate this limitation, consider the following simple example in Figure 5-1. In the polyhedral model, this triangular iteration domain is represented exactly by the following set of inequalities:

$$
\begin{aligned}
i &\geq 0 \\
-i + 5 &\geq 0 \\
i + j &\geq 0 \\
j &\geq 0 \\
-j + 5 &\geq 0
\end{aligned}
$$

In Halide, however, this triangular iteration domain is approximated with a more conservative axis-aligned bounding box: $0 \leq i \leq 5$ and $0 \leq j \leq 5$.

This limitation is particularly critical in distributed system, where a sender has to identify the region required to be sent to a receiver. If the maximum values of i and j had been 100 instead of 5, the sender would have needed to send significantly more data to the receiver which will never be used by the receiver.

NFM provides Halide the capability to represent iteration domain more precisely without sacrificing the capability to handle parametric tiling. In fact, NFM is able to handle more general cases of parametric tiling, not limited to rectangular tiling. As shown in Chapter 4, NFM can handle various tiling schemes, such as multi-level tiling and diamond tiling.

In addition, NFM provides a more general, unified, robust framework to perform computations on iteration domains (e.g merge, intersection, etc.) and to simplify expressions in a very simple way. In Halide, we need to write hundreds of rules to perform the simplifications. The `simplify` method in Halide is essentially a giant if/else statement handling those different simplification rules (see Figure A.1 in Appendix A).

5.2.1 Box Merging

We integrate NFM to simplify the box domain generated during `merge_boxes` method in Halide. We will refer to the NFM integrated method as `merge_boxes_nfm` and to the original Halide method as `merge_boxes_halide`. Some of the examples of simplified interval bound expressions by `merge_boxes_nfm` and `merge_boxes_halide` are shown in Table 5.1. The expression listed on the table is either the upper or

lower bound of an interval dimension of the merged box. During simplification, `merge_boxes_nfm` first converts the input interval expression into a union of polyhedral domains. A domain in the polyhedral model represents a conjunction (AND) of constraints, while a union domain represents a disjunction (OR) of domains. After simplification is performed via NFM in the polyhedral domain, `merge_boxes_nfm` converts the union domain back into Halide's interval bound expressions. Within a polyhedral domain, the upper bound of an interval is simply the minimum of all the upper bound constraints, e.g. $x \leq min(a, b)$. Likewise, the lower bound of an interval is simply the maximum of all the upper bound constraints, e.g. $x \geq max(a, b)$. Since Halide's iteration set is constrained to axis-aligned bounding boxes, a union domain in the polyhedral model is converted into either a *max* of the composing domains' lower bound expressions or a *min* of the composing domains' upper bound expressions.

To illustrate the concept, consider the following simple example of upper bound on w:

$$w \leq max(min(N, N+1),\ N+2) \tag{5.1}$$

where w is the loop dimension and N is a symbolic constant. We convert 5.1 into set of inequality constraints to be processed by NFM (Equation 5.2-5.4). The union domain representation of this expression is the following:

$$w \leq max(min(N,\ N+1),\ N+2) \tag{5.2}$$

$$\implies (w \leq min(N,\ N+1)) \quad \vee \quad (w \leq N+2) \tag{5.3}$$

$$\implies (w \leq N\ \wedge\ w \leq N+1) \quad \vee \quad (w \leq N+2) \tag{5.4}$$

$(w \leq N\ \wedge\ w \leq N+1)$ (Equation 5.4) is the first domain, while the $(w \leq N+2)$ (Equation 5.4) is the second domain in the union domain. NFM will detect that the constraint $w \leq N+1$ is redundant in the presence of $w \leq N$ in the first domain, and

hence will remove it from the domain. The final union domain is then the following:

$$(w \leq N) \quad \vee \quad (w \leq N + 2) \tag{5.5}$$

The conversion of equation 5.5 to Halide's interval bound is equal to the following:

$$w \leq max(N,\ N + 2) \tag{5.6}$$

We further use one of the functionalities provided by NFM to simplify $w \leq max(N,\ N+2)$ into $w \leq N + 2$. Thus, the final interval's upper bound on w is $N + 2$. Note that running Halide's `simplify` on equation 5.1 will also give the same answer as NFM. However, Halide's `simplify` is limited; it cannot simplify a more complex example like $min(min(p, min(q, min(p-2, 0))), p-1)$, where p and q are symbolic constants, as shown in Table 5.1. For Halide to handle this case, we need to add more rules into its `simplify` method.

halide	$max(min(p + q * 16 + r + s + t * 16 - 15, p + r + u - 14), min((q + t) * 16 + s - 16, u - 15) + p + r + 1)$
nfm	$min(p + q * 16 + r + s + t * 16 - 15, p + r + u - 14))$
halide	$max(min(((max((max(a, 1) - 1), 0) * 16) + q + 15), p), r)$
nfm	$max(r, min(max(q + 15, q + a * 16 - 1), p))$
halide	$min(min(p, min(q, min(p - 2, 0))), p - 1)$
nfm	$min(q, min(p - 2, 0))$
halide	$min(min(p, min(q, min(r, min(p-1, min(p-2, 0))))), min(q, min(r, s)))$
nfm	$min(r, min(q, min((p - 2), min(s, 0))))$

Table 5.1: Expressions simplified by `merge_boxes_halide` vs. `merge_boxes_nfm`. a, p, q, r, s, t, and u are symbolic constants.

5.2.2 Boxes Intersection

NFM provides a natural way to perform domain intersection. In the polyhedral model, intersection of two domains is equivalent to combining the two domains. For example,

the intersection of domain 1, $w \leq a$, and domain 2, $w \leq b$, is simply $w \leq a \ \wedge \ w \leq b$. NFM is also able to give comparable results, if not sometimes better than Halide.

Consider the two 1-D Halide boxes in Figure 5-2, where a, b, and M are symbolic constants; and x is the box dimension:

```
Box A:
    x: [10, 19]
Box B:
    x: [min(max(((M*a) + b), 0), 19), max(min((((M + 1)*a)
       + b), 19), 0))]
```

Figure 5-2: Example of 1-D Halide boxes

The intersections of those two boxes using NFM and Halide are shown in Figure 5-3. As we can see from the result, NFM result is simpler than Halide's.

```
Box Halide:
   x: [max(min(max(((M*a) + b), 0), 19), 10),
      min(max(min((((M + 1)*a) + b), 19), 0), 19)]
Box NFM:
    x: [min(max((b + (M*a)), 10), 19), max(min(((b + a) +
       (M*a)), 19), 0)]
```

Figure 5-3: Intersection of two boxes specified in Figure 5-2 using NFM and Halide respectively

5.3 Experimental Results

We ran both `merge_boxes_nfm` and `merge_boxes_halide` on several application examples provided in the Halide distribution. As performance measure, we count the total number of CPU cycles required to execute the generated expression. In this case, the expression is the min/max value of each box interval dimension. We choose number of CPU cycles instead of running time to measure the performance since the difference in running time is not easily observable. Comparing the expression length does not make any sense either, as shorter expression does not necessarily equal to

better performance; for example, $x + y + z$ is longer than x/y, however, x/y is more expensive since division takes longer CPU cycles to compute than addition.

We estimate the number of CPU cycles by summing the CPU latency of all instructions within an expression. The number of CPU cycles per instruction we used to measure the performance is listed in Table 5.2. The results of running `merge_boxes_nfm` and `merge_boxes_halide` on the applications are listed in Table 5.3. On average, the total number CPU cycles for executing `merge_boxes_nfm` is 79.04% of that of `merge_boxes_halide`.

As we can see from Table 5.3, `merge_boxes_nfm` performs better compared to `merge_boxes_halide` (`merge_boxes_nfm` has lower total number of CPU cycles), except for `linear_algebra`. In `linear_algebra`, `merge_boxes_nfm` does not perform as well as `merge_boxes_halide` because of a limitation in the NFM to Halide converter, particularly when converting NFM back into Halide's `select` expression. To make this point clearer, let's consider the example in Figure 5-4.

```
select(M >= 0, select(N >= 0, a, b), c)
```

Figure 5-4: Example of Halide's `select` expression

In Halide, `select(true, true_condition, false_condition)` is equivalent to if/else statement in C++. From this point onward, we shall refer to `select` as if/else. The expression in Figure 5-4 is equivalent to the expression in Figure 5-5 in C++.

```
if (M >= 0) {
  if (N >= 0) {
    a
  } else {
    b
  }
} else {
  c
}
```

Figure 5-5: C++ equivalent of expression in Figure 5-4

Converting the expression in Figure 5-5 into NFM's union domain, we get the

following:

$$\{(M \geq 0) \wedge (N \geq 0) \wedge a\} \vee \{(M \geq 0) \wedge (N < 0) \wedge b\} \vee \{(M < 0) \vee c\} \quad (5.7)$$

Assuming that Union Domain 5.7 is already in the simplest form, there are several equivalent if/else forms of Union Domain 5.7. One of them is listed in Figure 5-6.

```
if (M >= 0 && N >= 0) {
   a
} else if (M >= 0 && N < 0) {
   b
} else {
   c
}
```

Figure 5-6: C++ equivalent of expression in Figure 5-4

Comparing to the original expression in Figure 5-5, we now count the CPU cycles latencies of both M and N twice. With a better converter, we could have combined the first and the second statements in Figure 5-6 into a nested if/else statement instead, as in the original expression, which will cut down the total CPU cycles latencies.

Instruction	**# of CPU cycles**
CAST	1
ADD	1
SUB	1
MUL	3
DIV	24
MOD	1
RETURN	1
COMPARE	2
AND	1
OR	1
NOT	1
CALL	3

Table 5.2: Latency (Number of CPU cycles) of each instruction used to compare the performance of NFM with that of Halide

Application	**Halide (# of CPU cycles)**	**NFM (# of CPU Cycles)**	**NFM/Halide**
blur	432	370	85.65%
linear_algebra	34122	35276	103.38%
resize	7835	722	9.22%
fft	398693	340932	85.51%
local_laplacian	64922	50702	78.10%
camera_pipe	99833	92274	92.43%
wavelet	542	375	69.19%
interpolate	561664	399942	71.21%
bilateral_grid	1980	1697	85.71%

Table 5.3: Number of CPU cycles of interval expressions generated by `merge_box` when using NFM vs. using Halide

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 6

Conclusions

We have extended the Fourier-Motzkin elimination method to handle symbolic constant coefficients and built a framework, called NFM, that implements the extension. We have shown through a number of examples that extending the Fourier-Motzkin elimination allows us to solve for more complex code generation problems, such as parametric tiling, parallelization, and communication. We have also introduced some simplification steps to make the system more manageable to solve. We believe that this simple extension enables us to deal with common non-linear systems robustly, as demonstrated by the examples.

We have further integrated our NFM framework into Halide, an open-source domain-specific language compiler for image processing [13], which provides Halide a more robust framework to perform computation on iteration domain such as merge, intersection, etc., and provides a significant performance gain during loop bound computation. NFM is able to generate a more simplified expression than Halide, which translates to a lower CPU latency. NFM also provides Halide a unified framework to perform more complex optimization schemes, such as diamond tiling.

Given the promising results, the next steps will involve building a better NFM to Halide converter. As we see in Chapter 5.3, conversion of NFM's union domain back to Halide's `select` remains a problem. Simplifying an if/else statement into a nested one might require some type of heuristics to guide the simplification process. Another major step would be to modify Halide to use polyhedral representation for

its domain. Currently to use the NFM framework inside Halide, we need to convert Halide's expression into NFM's union domain and back into Halide's expression, which may potentially include a significant overhead to the compilation process. Converting NFM's union domain back into Halide's expression also means sacrificing the more precise domain representation of NFM since Halide can only represent an axis-aligned box domain.

In terms of the NFM framework, it may be beneficial to integrate ISL (Integer Set Library), a C library for manipulating integer sets bounded by affine constraints [16], into NFM. ISL has a relatively stable code base and has many functions which may be useful for further development of NFM. NFM could potentially use some of the functions already available ISL without having to reinvent the wheel.

Appendix A

Figures

Program A.1: Code snippet of Halide's `simplify min` method

```
void visit(const Min *op) {
    ...
    if (no_overflow(op->type) &&
        add_a &&
        const_int(add_a->b, &ia) &&
        add_b &&
        const_int(add_b->b, &ib) &&
        equal(add_a->a, add_b->a)) {
        // min(x + 3, x - 2) -> x - 2
        if (ia > ib) {
            expr = b;
        } else {
            expr = a;
        }
    } else if (no_overflow(op->type) &&
               add_a &&
               const_int(add_a->b, &ia) &&
               equal(add_a->a, b)) {
        // min(x + 5, x) -> x
```

```
    if (ia > 0) {
        expr = b;
    } else {
        expr = a;
    }
} else if (no_overflow(op->type) &&
           add_b &&
           const_int(add_b->b, &ib) &&
           equal(add_b->a, a)) {
    // min(x, x + 5) -> x
    if (ib > 0) {
        expr = a;
    } else {
        expr = b;
    }
} else if (no_overflow(op->type) &&
           sub_a &&
           sub_b &&
           equal(sub_a->b, sub_b->b) &&
           const_int(sub_a->a, &ia) &&
           const_int(sub_b->a, &ib)) {
    // min (100-x, 101-x) -> 100-x
    if (ia < ib) {
        expr = a;
    } else {
        expr = b;
    }
} else if (a_round_up.defined() &&
           equal(a_round_up, b)) {
    // min(((a + 3)/4)*4, a) -> a
    expr = b;
} else if (a_round_up.defined() &&
```

```
            max_b &&
            equal(a_round_up, max_b->a) &&
            is_const(max_b->b, a_round_up_factor)) {
    // min(((a + 3)/4)*4, max(a, 4)) -> max(a, 4)
    expr = b;
} else if (b_round_up.defined() &&
            equal(b_round_up, a)) {
    // min(a, ((a + 3)/4)*4) -> a
    expr = a;
} else if (b_round_up.defined() &&
            max_a &&
            equal(b_round_up, max_a->a) &&
            is_const(max_a->b, b_round_up_factor)) {
    // min(max(a, 4), ((a + 3)/4)*4) -> max(a, 4)
    expr = a;
} else if (max_a &&
            equal(max_a->b, b)) {
    // min(max(x, y), y) -> y
    expr = b;
} else if (min_a &&
            (equal(min_a->b, b) || equal(min_a->a, b))) {
    // min(min(x, y), y) -> min(x, y)
    expr = a;
} else if (min_b &&
            (equal(min_b->b, a) || equal(min_b->a, a))) {
    // min(y, min(x, y)) -> min(x, y)
    expr = b;
} else if (min_a &&
            min_a_a &&
            equal(min_a_a->b, b)) {
    // min(min(min(x, y), z), y) -> min(min(x, y), z)
    expr = a;
```

```
} else if (min_a &&
           min_a_a_a &&
           equal(min_a_a_a->b, b)) {
    // min(min(min(min(x, y), z), w), y) -> min(min(min(x,
       y), z), w)
    expr = a;
} else if (min_a &&
           min_a_a_a_a &&
           equal(min_a_a_a_a->b, b)) {
    // min(min(min(min(min(x, y), z), w), l), y) ->
       min(min(min(min(x, y), z), w), l)
    expr = a;
} else if (max_a &&
           max_b &&
           equal(max_a->a, max_b->a)) {
    // Distributive law for min/max
    // min(max(x, y), max(x, z)) -> max(min(y, z), x)
    expr = mutate(Max::make(Min::make(max_a->b, max_b->b),
       max_a->a));
} else if (max_a &&
           max_b &&
           equal(max_a->a, max_b->b)) {
    // min(max(x, y), max(z, x)) -> max(min(y, z), x)
    expr = mutate(Max::make(Min::make(max_a->b, max_b->a),
       max_a->a));
} else if (max_a &&
           max_b &&
           equal(max_a->b, max_b->a)) {
    // min(max(y, x), max(x, z)) -> max(min(y, z), x)
    expr = mutate(Max::make(Min::make(max_a->a, max_b->b),
       max_a->b));
} else if (max_a &&
```

```
            max_b &&
            equal(max_a->b, max_b->b)) {
    // min(max(y, x), max(z, x)) -> max(min(y, z), x)
    expr = mutate(Max::make(Min::make(max_a->a, max_b->a),
       max_a->b));
} else if (min_a &&
            min_b &&
            equal(min_a->a, min_b->a)) {
    // min(min(x, y), min(x, z)) -> min(min(y, z), x)
    expr = mutate(Min::make(Min::make(min_a->b, min_b->b),
       min_a->a));
} else if (min_a &&
            min_b &&
            equal(min_a->a, min_b->b)) {
    // min(min(x, y), min(z, x)) -> min(min(y, z), x)
    expr = mutate(Min::make(Min::make(min_a->b, min_b->a),
       min_a->a));
} else if (min_a &&
            min_b &&
            equal(min_a->b, min_b->a)) {
    // min(min(y, x), min(x, z)) -> min(min(y, z), x)
    expr = mutate(Min::make(Min::make(min_a->a, min_b->b),
       min_a->b));
} else if (min_a &&
            min_b &&
            equal(min_a->b, min_b->b)) {
    // min(min(y, x), min(z, x)) -> min(min(y, z), x)
    expr = mutate(Min::make(Min::make(min_a->a, min_b->a),
       min_a->b));
} else if (no_overflow(op->type) &&
            add_a &&
            add_b &&
```

```
            equal(add_a->b, add_b->b)) {
    // Distributive law for addition
    // min(a + b, c + b) -> min(a, c) + b
    expr = mutate(min(add_a->a, add_b->a)) + add_a->b;
} else if (no_overflow(op->type) &&
            add_a &&
            add_b &&
            equal(add_a->a, add_b->a)) {
    // min(b + a, b + c) -> min(a, c) + b
    expr = mutate(min(add_a->b, add_b->b)) + add_a->a;
} else if (no_overflow(op->type) &&
            add_a &&
            add_b &&
            equal(add_a->a, add_b->b)) {
    // min(b + a, c + b) -> min(a, c) + b
    expr = mutate(min(add_a->b, add_b->a)) + add_a->a;
} else if (no_overflow(op->type) &&
            add_a &&
            add_b &&
            equal(add_a->b, add_b->a)) {
    // min(a + b, b + c) -> min(a, c) + b
    expr = mutate(min(add_a->a, add_b->b)) + add_a->b;
} else if (min_a &&
            is_simple_const(min_a->b)) {
    if (is_simple_const(b)) {
        // min(min(x, 4), 5) -> min(x, 4)
        expr = Min::make(min_a->a, mutate(Min::make(b,
           min_a->b)));
    } else {
        // min(min(x, 4), y) -> min(min(x, y), 4)
        expr = mutate(Min::make(Min::make(min_a->a, b),
           min_a->b));
```

```
            }
        }
        ...
}
```

THIS PAGE INTENTIONALLY LEFT BLANK

Bibliography

[1] Saman P. Amarasinghe and Monica S. Lam. Communication optimization and code generation for distributed memory machines. *SIGPLAN Not.*, 28(6):126–138, June 1993.

[2] Saman Prabhath Amarasinghe. *Parallelizing Compiler Techniques Based on Linear Inequalities*. PhD thesis, Stanford University, 1997.

[3] Vinayaka Bandishti, Irshad Pananilath, and Uday Bondhugula. Tiling stencil computations to maximize parallelism. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 40:1–40:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[4] Cedric Bastoul. Code generation in the polyhedral model is easier than you think. In *In IEEE Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT 04)*, pages 7–16, 2004.

[5] Uday Bondhugula, J. Ramanujam, and et al. Pluto: A practical and fully automatic polyhedral program optimization system. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI 08)*, 2008.

[6] Geir Dahl. Combinatorial properties of fourier-motzkin elimination. *Electronic Journal of Linear Algebra*, 16, 2007.

[7] Paul Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20, 1991.

[8] Armin Größlinger. Extending the polyhedron model to inequality systems with non-linear parameters using quantifier elimination. Master's thesis, University of Passau, 2003.

[9] Francois Irigoin and Remi Triolet. Supernode partitioning. In *Symposium on Principles of Programming Languages (POPL'88)*, pages 319–328, San Diego, CA, January 1988.

[10] Richard M Karp, Raymond E Miller, and Shmuel Winograd. The organization of computations for uniform recurrence equations. *Journal of the ACM*, 14(3):563–590, 1967.

[11] Jonathan Ragan-Kelley. *Decoupling Algorithms from the Organization of Computation for High Performance Image Processing.* PhD thesis, Massachusetts Institute of Technology, June 2014.

[12] Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Trans. Graph.*, 31(4):32:1–32:12, July 2012.

[13] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 519–530, New York, NY, USA, 2013. ACM.

[14] Lakshminarayanan Renganarayanan, DaeGon Kim, Sanjay Rajopadhye, and Michelle Mills Strout. Parameterized tiled loops for free. *SIGPLAN Not.*, 42(6):405–414, June 2007.

[15] Sanket Tavarageri, Albert Hartono, Muthu Baskaran, Louis-Noël Pouchet, J. Ramanujam, and P. Sadayappan. Parametric tiling of affine loop nests. In *15th Workshop on Compilers for Parallel Computing (CPC'10)*, Vienna, Austria, July 2010.

[16] Sven Verdoolaege. Isl: An integer set library for the polyhedral model. In *Proceedings of the Third International Congress Conference on Mathematical Software*, ICMS'10, pages 299–302, Berlin, Heidelberg, 2010. Springer-Verlag.