McFLAT : A PROFILE-BASED FRAMEWORK FOR MATLAB LOOP ANALYSIS AND TRANSFORMATIONS

by

Amina Aslam

School of Computer Science
McGill University, Montréal

August 2010

A THESIS SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

Abstract

Parallelization and optimization of the MATLAB® programming language presents several challenges due to the dynamic nature of MATLAB. Since MATLAB does not have static type declarations, neither the shape and size of arrays, nor the loop bounds are known at compile-time. This means that many standard array dependence tests and associated transformations cannot be applied straight-forwardly. On the other hand, many MATLAB programs operate on arrays using loops and thus are ideal candidates for loop transformations and possibly loop vectorization/parallelization.

This thesis presents a new framework, McFLAT, which uses profile-based training runs to determine likely loop-bounds ranges for which specialized versions of the loops may be generated. The main idea is to collect information about observed loop bounds and hot loops using training data which is then used to heuristically decide upon which loops and which ranges are worth specializing using a variety of loop transformations.

Our McFLAT framework has been implemented as part of the McLAB extensible compiler toolkit. Currently, McFLAT is used to automatically transform ordinary MATLAB code into specialized MATLAB code with transformations applied to it. This specialized code can be executed on any MATLAB system, and we report results for four execution engines, Mathwork's proprietary MATLAB system, the GNU Octave open-source interpreter, McLAB's McVM interpreter and the McVM JIT. For several benchmarks, we observed significant speedups for the specialized versions, and noted that loop transformations had different impacts depending on the loop range and execution engine.

This thesis reports on the design and implementation of McFLAT, a framework that is designed to study the effect of various loop transformations on different loop-bound ranges

by introducing loop-level specializations in MATLAB programs.

Résumé

La parallelisation et l′optimisation du langage de programmation informatique MATLAB® représente plusieurs défis compte tenu du caractère dynamique de ce dernier.

Puisque MATLAB ne possède pas de déclaration de type statique, ni son profil et la taille des matrices ni les boucles limites, sont connus au temps de compilation. Cela signifie que plusieurs tests standardisés de la dépendance des matrices et de ses transformations associées ne peuvent pas *ê*tre appliqués dans une manière directe. D ′autre part, plusieurs programmes MATLAB sont opérés sur les matrices en utilisant les boucles et donc sont des candidats idéals pour la transformation en boucle et possiblement la vectorisation en boucle/parallelisation.

Cette hypothèse présente un nouveau cadre, McFLAT, qui exécute des entrainements bases sur des profils afin de déterminer la portée des éventuelles boucles-limites pour qui des versions spécialisées des boucles pourraient *ê*tre générées. L′idée principale est de faire une collecte d′information concernant les boucles en observation ainsi que les boucles chauds en capitalisant sur les données d′entrainement qui sont ensuite utilisées pour décider heuristiquement sur quels boucles et limites il faut se spécialiser en utilisant une variété de transformateurs de boucles.

Notre cadre McFLAT a été implémenté en tant que composant de McLAB extensible compiler toolkit. Actuellement, McFLAT, est utilisé pour transformer automatiquement le code ordinaire MATLAB en code spécialisé MATLAB avec des transformations appliquées à ce dernier. Ce code spécialisé pourrait ensuite *ê*tre exécuté sur tout systéme matlab et nous livrons les résultats de quatre moteurs d′exécution, Mathwork's proprietary MATLAB system, le GNU Octave source-libre interprète, l′interprète McVM de mclab et le McVM

JIT. Pour plusieurs repères, nous observons une rapidité significative pour les versions spécialisées, et on note que les transformations en boucle ont eu différents impacts qui dépendent de la limite des boucles et du moteur d′ exécution.

Ce mémoire se focalise sur le dessin et l′implémentation de McFLAT, un cadre qui est désigné pour étudier les effets de plusieurs transformations en boucle sur plusieurs niveaux de boucles-limites en introduisant des spécialisations au niveau des boucles dans les programmes MATLAB.

Acknowledgements

This work was supported, in part, by the Natural Sciences and Engineering Research Council of Canada (NSERC).

I am thankful to my supervisor, Laurie Hendren, whose encouragement, guidance and support from the initial to the final level enabled me to develop an understanding of the problem.

I would like to thank all the members of the McLAB team for providing the basis which McFLAT uses.

I would also like to thank my parents, siblings and my friend Bilal Ahmad for always encouraging me and believing in me.

Table of Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

MATLAB is a high-level, untyped and interpreted language which is commonly used by scientists and engineers because of its ease of use, high-level syntax for arrays, and a rich collection of built-in library functions. Scientific programs often use a collection of loops, which may be in sequence and/or nested to perform tasks multiple times. Most execution time in scientific programs is spent in loops [Moo06]. MATLAB is not necessarily optimized for the use of loops and some of the slowest blocks of code that inflate MATLAB program execution time are `for/while` loops. Therefore, compiler analysis and loop optimization techniques are required to make the execution of loops faster and to introduce parallelism so as to take advantage of multi-core CPUs and GPGPU(General Purpose Graphic Processing Unit) computing capabilities.

Recently, there has been a tremendous increase in the popularity of dynamic languages such as Python, Ruby, PHP, JavaScript and MATLAB. These languages are developed with programmer convenience in mind. This ease of use comes with a price, which is, poor execution performance as compared to statically-compiled languages (e.g.: C, C++, Fortran, etc.).

McFLAT, A framework for loop Analysis and Transformations is a component of a larger effort known as the McLAB project[1], being developed by Sable Lab at McGill Univer-

[1] www.sable.mcgill.ca/mclab

sity. The overall goal of McLAB is to find ways to improve the performance, usefulness and accessibility of current scientific programming languages. The McLAB team currently focuses its efforts on the MATLAB programming language and its extensions e.g. AspectMatlab [TAH10].

Tuning a critical loop in a scientific program can lead to a significant reduction in its execution time. The dynamic nature of MATLAB programming language poses several challenges for optimizations including loop transformations and loop vectorization/parallelization. Since MATLAB does not have static type declarations, neither the shape and size of arrays, nor the loop bounds are known at compile-time. This means that many standard array dependence tests and associated transformations cannot be applied straight-forwardly.

This thesis presents a new framework, McFLAT, which uses profile-based training runs to determine likely loop-bounds ranges for which specialized versions of the loops may be generated. The main idea is to collect information required for loop optimizations using training data and then decide heuristically which loop bounds are important. The resulting output is a set of programs with valid loop transformations applied for important predicted loop-bounds ranges.

1.1 Contributions

The McFLAT project makes the following contributions:

- Design and implementation of the *Profiler* which instruments the MATLAB programs and collects information about the loop bounds and program features.
- Design and implementation of the *Range Estimator* that predicts important loop-bound ranges based on profiled information.
- Implementation of a set of efficient dependence testers to determine dependence between same array accesses for important predicted ranges. This component determines whether loop transformation(s) can be applied or not.

- Design and implementation of the legality tester that determines which loop transformation(s) and in what order should be applied.
- Design and implementation of the *Loop Transformer* that introduces legal or programmer-suggested loop transformations into the MATLAB code for important predicted ranges.
- Design and implementation of an automatic parallelization detection mechanism in the context of the MATLAB programming language.
- A detailed analysis of the impact of loop transformations on different loop-bounds ranges and different execution engines. i.e. Mathwork's proprietary MATLAB system, the GNU Octave open-source interpreter, McLAB's McVM interpreter and the McVM JIT(McJIT).

The McFLAT framework, has been implemented as part of the McLAB extensible compiler toolkit. Currently, McFLAT is used to automatically transform ordinary MATLAB code into specialized MATLAB code with valid loop transformations applied for important predicted loop-bounds ranges. This specialized code can be executed on any MATLAB system, and we report results for four execution engines, Mathwork's proprietary MATLAB system, the GNU Octave open-source interpreter, McLAB's McVM interpreter and the McVM JIT(McJIT). For several benchmarks, we observed significant performance speedups for the specialized versions, and noted that loop transformations had different impacts depending on the loop range, execution engine and the source program features.

1.2 Thesis Outline

This thesis is divided into 6 chapters (including this introduction chapter). *Chapter 2* gives a general overview of MATLAB programming language, its features and execution model. It then discusses the McLAB project, its various components and how McFLAT fits into the overall picture. In *Chapter 3* we present the overall architecture of the McFLAT framework and its different phases in detail. *Chapter 4* reports results for four execution engines, Mathwork's proprietary MATLAB system, the GNU Octave open-source interpreter,

McLAB's McVM interpreter and the McVM JIT(McJIT) and discusses the impact of loop transformations on different loop-bound ranges and different execution engine. *Chapter 5* discusses related work done in the context of this thesis which helped us to form the base of our research, and the ways in which our approach differs with them. Finally, *Chapter 6* presents our conclusions and outlines some possible future research work in this domain.

Chapter 2

Background

In this chapter we present background information helpful to the understanding of this thesis. We begin with a brief overview of the MATLAB programming language, its importance for the scientific and engineering community, and its execution environment. This is followed by a discussion on MATLAB `for` loop construct, arrays indexing and array operations, and a programming code example.

This is followed by a discussion of dynamic languages and the associated optimization challenges particularly in the context of loop transformations. We then present an overview of the McLAB project talking about its various components and where does McFLAT fits into the overall picture.

2.1 The MATLAB Language

MATLAB is a numerical computing environment, originally invented in the late 1970s by Cleve Moler, then a professor of computer science at the University of New Mexico. He designed the language to give his students access to some of the power of FORTRAN, without having to learn the FORTRAN language itself [Matb]. Developed for providing an easier numerical computing environment to students, MATLAB offered flexible syntactic constructs, which also made it popular amongst other computationally-intensive research

areas. Since then, MATLAB is widely used in the academic, scientific and engineering communities.

MATLAB is a very popular technical computing environment and a fourth generation programming language. It is dynamically-typed, weakly-typed procedural language. The name MATLAB stands for *MATrix LABoratory*, because its basic data element is a matrix. MATLAB provides a large library of common matrix operations (e.g.: addition, inversion, multiplication). In addition to its matrix orientation MATLAB incorporates many features found in other dynamic languages, such as the runtime creation of closures. It also allows implementation of algorithms, plotting of functions and data, creation of user interfaces, and interfacing with programs written in other languages, including C, C++, and Fortran.

2.1.1 MATLAB's Execution Environment

In the MATLAB environment, we can write and execute programs, or scripts, that contain MATLAB commands, observe the results, and then execute another MATLAB command that interacts with the information already in the workspace. This interactive environment does not requires a formal compilation, linking/loading and execution process. However, errors in the syntax of MATLAB commands are detected when the MATLAB environment attempts to translate the command, and logical errors lead to execution errors when the MATLAB environment attempts to execute the command.

2.1.2 Supported Features

The Mathworks implementation of MATLAB offers a very rich feature set. The short list below enumerates some of its most used features [Mata, CB09]:

- Uniform treatment of all basic types as matrices
- Optimized built-in matrix operations
- Advanced graphic capabilities.

- Function handles, inline functions, feval (for function evaluation)
- Support for variety of industry standard file formats and other custom file formats
- Extensive library of numerical algorithms
- Creation of custom tool boxes
- Interactive mode with read-eval-print loop
- Code editor and debugging environment
- Effective documentation search system
- Built-in support for complex-numbers
- Repetition structures `for and while` loops
- Range expressions and array slicing/reshaping
- Nested function definitions
- Creation of closures from nested functions
- Creation of closures from lambda expressions
- Graphical 2D and 3D plotting tools
- C and FORTRAN function wrapping

MATLAB has evolved tremendously over the years. In addition to having added or modified several internal algorithms [Moo06], the MATLAB command interpreter now includes acceleration features, collectively called *The* MATLAB *JIT-Accelerator* [Mat02]. The accelerator has increased the speed of loop operations by interpreting and executing code within a loop as a whole, rather than line by line. However, to make use of JIT Accelerator, loop operation code must follow specific guidelines, if these guidelines are not followed, loop operation code is interpreted at a much slower line-by-line rate [Moo06].

MATLAB code having loops, benefit from MATLAB's *JIT-Accelerator*, if it has the following properties [DB05].

- Only `for` loops are optimized.
- The loop should contain specific logical, character string, double-precision, and less than 64-bit integer data types.
- The loops can only use arrays having three or less dimensions.
- All variables used within a loop must be defined prior to loop execution.
- Memory for all variables within a loop is preallocated, and all the variables should maintain constant size and data type throughout the execution of the loop.
- Loop indices must be scalar quantities.
- Only built-in MATLAB functions can be called with-in a loop.
- Conditional statements with *if-then-else* or *switch-case* constructions should complex conditions, only scalar comparisons are optimized.

2.1.3 MATLAB's Control Flow

Computer programming languages offer features that allow you to control the flow of command execution using decision making structures. Control flow is very important since it lets past computations influence future operations. MATLAB offers five decision making or control flow structures.

- `for` Loops.
- `while` Loops.
- `if-else-end` construction.
- `switch-case` construction.
- `try-catch` block.

2.1.4 For Loops

In `for` loops the execution of a command or a group of commands is repeated a predetermined number of times. MATLAB provides following different syntax for `for` loop which are as follows.

2.1.5 Syntax

for *y= initval: endval, statements,* end

for *y= initval: stepval: endval, statements,* end

for *y= arr, loop statements,* end

2.1.6 Description

The construct " *for y=initval: endval, statements,* end ", repeatedly executes one or more MATLAB statements in a loop. The loop counter variable *y* is initialized to value *initval* at the start of the first pass through the loop, and automatically increments by 1 each time through the loop. The program repeatedly iterates through statements until either *y* has incremented to the value *endval*, or MATLAB encounters a *break*, or *return* instruction, thus forcing an immediate exit of the loop. If MATLAB encounters a *continue* statement in the loop body, it immediately exits the current iteration at the location of the *continue* statement, skipping any remaining code in that iteration, and begins another iteration at the start of the loop statements with the value of the loop counter incremented by one.

The " *for y= arr loop statements,* end ", this `for` statement on each iteration creates a column vector *index* from subsequent columns of array *arr*. For example, on the first iteration, the first column of array *arr* would be assigned to column vector index *index* = $array(:,1)$. The loop executes for a maximum of *n* times, where *n* is the number of columns of *arr*. In the case of *Listing 2.2*, where a row-vector is assigned to the index variable, the loop would iterate 6 times, as there are six columns in the row-vector and each time one column would be assigned to column vector index *index* and each column in this *Listing*

2.2 has one element.

The values *initval* and *endval* must be real numbers as in *Listing 2.1*, or arrays of real numbers as in *Listing 2.2*, or can also be calls to functions that return the same. The value assigned to *y* is often used in the code within the loop, however it is recommended that you do not assign to *y* in the loop code as in *Listing 2.3*.

The construct " *for y= initval: stepval: endval, statements*, end ", is the same as the above syntax, except that loop counter *y* is incremented (or decremented when *stepval* is negative) by the value *stepval* on each iteration through the loop. The value *stepval* must be a real number or can also be a call to a function that returns a real number.

```
1 for i=1:3:10
2    y=i^2;
3 end
```

Listing 2.1 A MATLAB `for` Loop Example

```
1 for i= [7,9,-1,3,3,5]
2    y=i*i;
3 end
```

Listing 2.2 A MATLAB `for` Loop Example with Specific Values Assigned from a Vector

```
1 for i=1:2
2   y=i^2;
3   i = 10;
4 end
```

Listing 2.3 A MATLAB `for` Loop Example with Loop Index Variable Re-assigned

McFLAT, handles the case where the `for` loop has a range expression defined, as the upper and lower bounds of the loop index variables are required to compute the dependence tests and apply transformations as in *Listing 2.1*.

2.1.7 Array and Array Operations in MATLAB

Arrays are a fundamental data structure that MATLAB uses to store and manipulate data. An array in the context of the MATLAB programming language is the same as in any other programming language, a list of numbers arranged in rows and/or columns. The simplest array(one-dimensional) is a row, or a column of numbers also called a *vector*. A more complex array (two-dimensional) is a collection of numbers arranged in rows and columns also called a *matrix*. There are also two container data types called *cell arrays* and *structures* which unlike arrays, allow grouping of dissimilar, but related, arrays into a single variable. Elements in an array can be addressed individually or in subgroups using subscripts. This is useful when there is a need to redefine only some of the elements of the array, to use specific elements in calculations, or when a subgroup of the elements is used to define a new variable, this capability is referred to as *slicing*. The colon operator is used to address a range of elements in arrays.

Listing 2.4 shows sample output of MATLAB being run in interactive mode. In this example, we show that the matrix variable *Mat* can be indexed by using subscripts, similar to the way two dimensional arrays are indexed in languages like Java (except that arrays in MATLAB are stored in column-major order).

```
>> % Here a matrix is being assigned to variable "Mat"
>> Mat = [10 20 30 40; 50 60 70 80; 90 100 110 120; 130 140 150 160]

Mat =

    10 20 30 40
    50 60 70 80
    90 100 110 120
   130 140 150 160

>> % Mat is accessed using scalar indexing, the last element of the
   first row is read
>> Mat(1,4)
```

```
13
14 ans =
15
16     40
17
18 >> % An entire row or column of Mat can be read by specifying a range
       of indices with the colon operator
19 >> Mat(:,1)
20
21 ans =
22
23     10
24     50
25     90
26    130
27
28 % One can also write to a sub-array (or slice) of Mat using ranges of
       indices
29 >> Mat(1, 2:3) = [7 7]
30
31 Mat =
32
33     10 7 7 4
34     50 60 70 80
35     90 100 110 120
36    130 140 150 160
```

Listing 2.4 Array Addressing or Indexing in MATLAB

2.1.8 MATLAB Code Examples

The example shown in *Listing 2.5* shows a MATLAB function that finds the Crank-Nicholson solution to the one-dimensional heat equation. This function could be invoked by inputting the `U = crnich(2.5, 1.5, 2, 321, 321);` command at the prompt, or it could be called from another function. This example demonstrates the use of a `for` loop, matrix multipli-

cation and exponentiation operators on line 72, array indexing using subscripts on lines 78 and 93 and array indexing using colon operator on line 96. Note that MATLAB array indices start from 1 instead of 0. Line 96 demonstrates the use of apostrophe transpose operator, which performs a complex conjugate transposition. MATLAB has two transpose operators. The apostrophe operator (e.g., B') performs a complex conjugate transposition. It flips a matrix about its main diagonal, and also changes the sign of the imaginary component of any complex elements of the matrix. The dot-apostrophe operator (e.g., $B.'$), transposes without affecting the sign of complex elements. The `Vd = s1*ones(1, n);` statement creates a row vector of size $1 \times n$ initialized with all ones. The row vector is then multiplied with a scalar $s1$.

```
1  function U = crnich(a, b, c, n, m)
2  %-----------------------------------------------------------------------
3  %
4  % This function M-file finds the Crank-Nicholson solution
5  % to the one-dimensional heat equation
6  %
7  %    2
8  %   u (x, t) = c u (x, t).
9  %   t xx
10 %
11 %  Invocation:
12 %  >> U = crnich(a, b, c, n, m)
13 %
14 %  where
15 %
16 %  i. a is the length of the metal rod,
17 %
18 %  i. b is the time duration,
19 %
20 %  i. c is the square root of the thermal
21 %  conductivity constant in the heat equation,
22 %
23 %  i. n is the number of grid points over [0, a],
```

```
%
%  i. m is the number of grid points over [0, b],
%
%  o. U is the solution matrix.
%
% Source:
%  Numerical Methods: MATLAB Programs,
%  (c) John H. Mathews, 1995.
%
% Author:
%  John H. Mathews (mathews@fullerton.edu).
%
% Date:
%  March 1995.
%
%-------------------------------------------------------------------------

h = a/(n-1);
k = b/(m-1);
r = c^2*k/h^2;
s1 = 2+2/r;
s2 = 2/r-2;
U = zeros(n, m); % initialize an n x m dimensional array U
for i1 = 2:(n-1)
   % call the built-in function sin and assigns the value to array U
   U(i1, 1) = sin(pi*h*(i1-1))+sin(3*pi*h*(i1-1));
end;

% creates a vector Vd with all of the values initialized to 1
Vd = s1*ones(1, n);
Vd(1) = 1;
Vd(n) = 1;
Va = -ones(1, n-1);
Va(n-1) = 0;
% creates a vector Vc with all the values initialized to -1
Vc = -ones(1, n-1);
Vc(1) = 0;
```

```
61 Vb(1) = 0;
62 Vb(n) = 0;
63
64 for j1 = 2:m
65    for i1 = 2:(n-1),
66   Vb(i1) = U(i1-1, j1-1)+U(i1+1, j1-1)+s2*U(i1, j1-1);
67    end;
68    % calls a user-defined function tridiagonal that returns a
         row-vector X
69    X = tridiagonal(Va, Vd, Vc, Vb);
70    % Takes a transpose of vector X and assigns the values to a subset
         of array U.
71    U(1:n, j1) = X';
72 end;
```

Listing 2.5 A MATLAB program example

2.1.9 Loop Optimization Challenges

Dynamic languages pose several optimization challenges due to their semantics. They are typically harder to optimize than their statically-compiled counterparts due to their dynamic nature. That is, the semantics of dynamic languages make it harder to predict their exact behavior at run-time.

MATLAB programs operate on arrays using loops and thus are good candidates for loop transformations and possible loop vectorization/parallelization. One of the challenges involved in applying loop optimizations in dynamic languages like MATLAB is that neither the shape and size of arrays, nor the loop bounds are known at compile-time [AH10]. This means that many standard array dependence tests, a pre-requisite, for applying loop transformations cannot be computed at compile-time.

To efficiently apply loop transformations in dynamic languages, optimizing compilers must find ways to improve performance of these languages without compromising the flexibility that they offer to the programmer. Profile-based techniques have been used in the past to suggest recompilation with additional optimizations.

2.2 The McLAB Framework

Dynamic languages are becoming increasingly popular. Common dynamic languages include Python, Perl, PHP, Ruby, Scheme, Smalltalk and MATLAB. These languages do not have static-type declarations, the variable types and their values, the size and shape of arrays and the loop bounds are unknown at compile time.

Recently, dynamic languages have started becoming more widely used due to the ease of use and flexibility, that these offer to programmers as compared to statically-compiled languages. Dynamic languages put fewer constraints on the programmer and they achieve more "work" per line of code and thus allow programmers to be more productive. Similarly, MATLAB, another dynamic language is widely used for computation intensive tasks. The combination of computational and visualization power makes it particularly useful for scientists and engineers.

MATLAB is a very popular programming language for technical computing used by students, engineers, and scientists in universities, research institutes, and industries all over the world. It was designed for sophisticated matrix and vector operations, which are common in scientific and engineering applications. It also offers a simple syntax that is familiar to most engineers and scientists. However, this ease of use comes with a price, MATLAB applications are generally slower than those programmed with statically-compiled languages. Keeping in mind the ease and fast development time that these languages offer, there is a strong desire in the compiler community to improve their performance.

The work presented in this thesis is a component of the McLAB framework [1]. The McLAB framework provides an extensible set of compilation, analysis and execution tools built around the core MATLAB programming language. One goal of the McLAB project is to provide an open-source set of tools for the programming languages and compiler community so that researchers (including our group) can develop new domain-specific language extensions and new compiler optimizations. A second goal is to provide engineers and scientists with these new languages and compilers which are more tailored to their needs

[1] www.sable.mcgill.ca/mclab

and also give better performance.

2.2.1 Overview

The overall structure of the McLAB framework is outlined in *Figure 2.1*.

The framework comprises of an extensible front-end, a high-level analyses engine, array dependence analysis and loop transformation framework (McFLAT, the topic of this thesis is represented with a shaded box) and three back-ends.

Currently, there is support for the core MATLAB language and also a complete extension supporting AspectMatlab[TAH10]. The front-end and the extensions are built using our group's extensible lexer, Metalexer[Cas09] and JastAdd [EH07]. There are three back-ends: McFor, a FORTRAN code generator [Li09]; a MATLAB generator (to use McLAB as a source-to-source compiler); and McVM, a virtual machine that includes a simple interpreter and a sophisticated type-specialization based JIT compiler(McJIT), which generates LLVM code [CBHV10].

In this chapter, we talked about the MATLAB programming language, its features, execution model and code example. We also discussed the McLAB framework and how McFLAT fits in the overall picture.

In *Chapter 3*, we discuss different components of the McFLAT framework in detail, in *Chapter 4*, we present experimental results obtained after applying our framework on a selection of benchmarks.

The McLab Framework

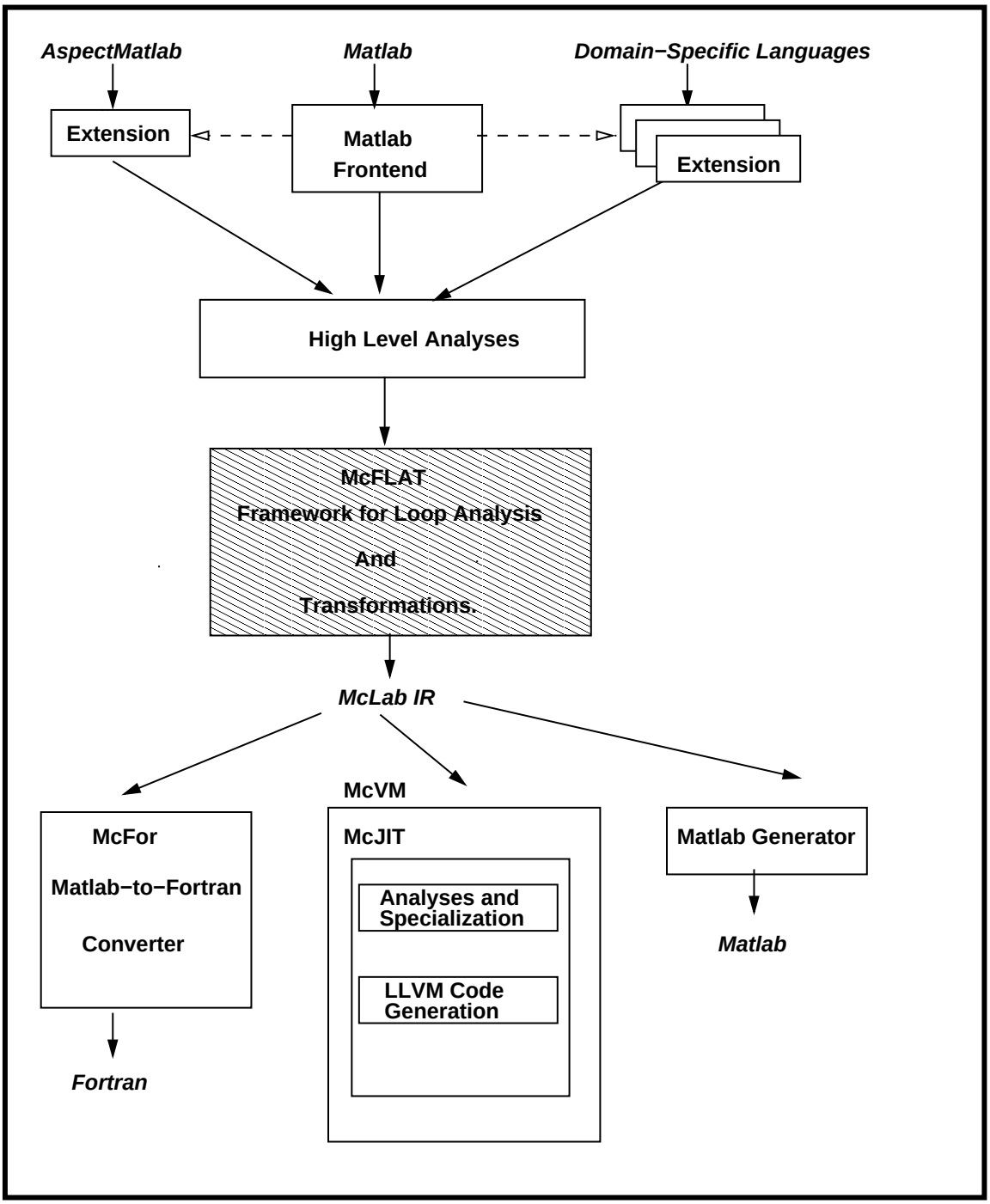

Figure 2.1 Structure Of The McLab Framework.

Chapter 3

McFLAT: A Framework for Loop Analysis and Transformations for MATLAB

3.1 Introduction

In this chapter, we provide a detailed overview of the key components of our McFLAT framework, and we also discuss parallel loop detection in the context of MATLAB programming language and some current limitations of the framework.

3.2 Overall Architecture

The overall structure of the McFLAT framework is outlined in *Figure 3.1*. Our ultimate goal is to embed this framework in our McJIT system, however currently it is a stand-alone source-to-source framework which uses the McLAB front-end. The user provides both the MATLAB program which they wish to optimize and a collection of representative inputs (top of *Figure 3.1*). The output of the system is a collection of specialized programs (bottom of *Figure 3.1*), where each specialized program has a different set of transformations

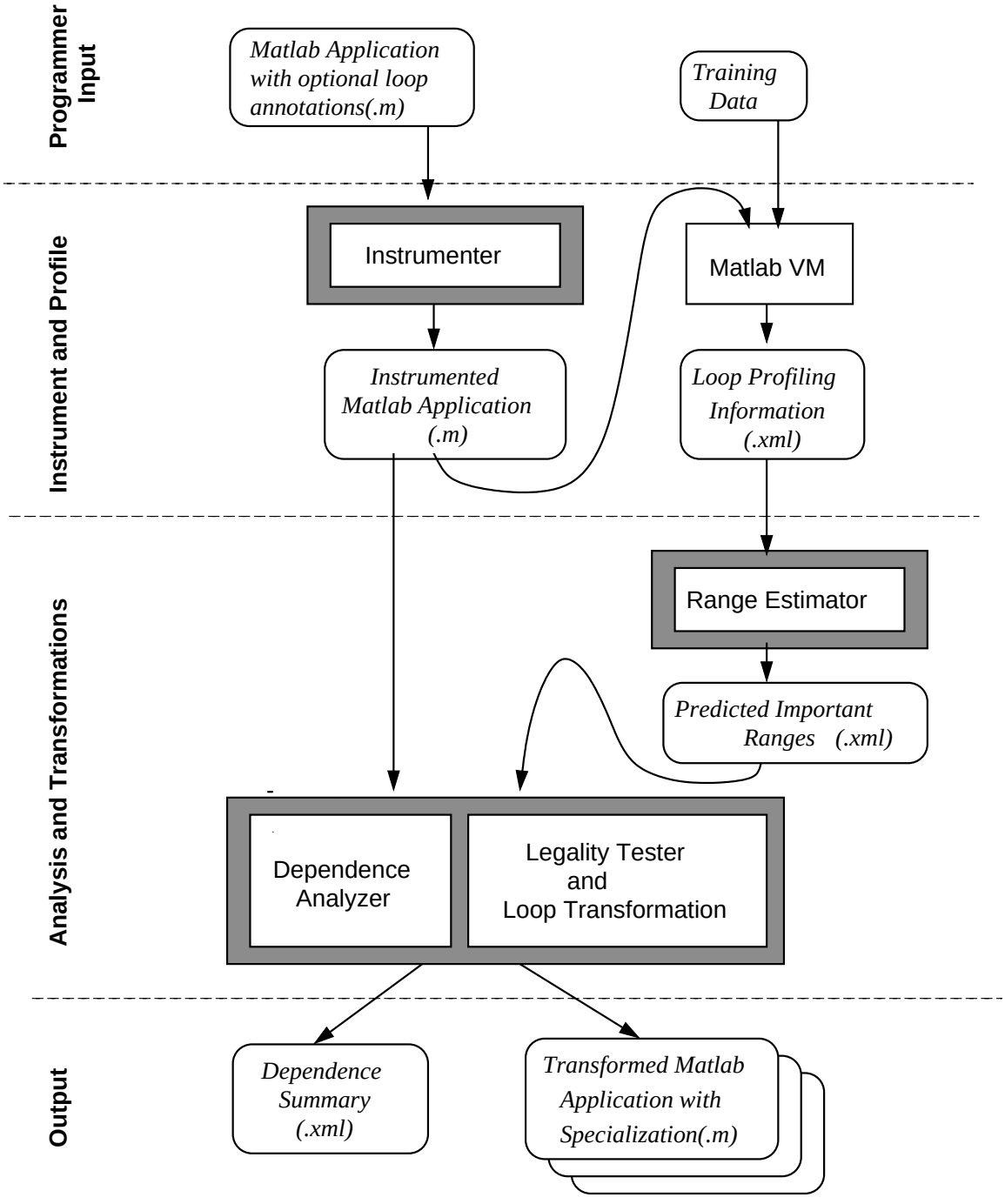

Figure 3.1 Structure of the McFLAT Framework.

applied. The system also outputs a dependence summary for each loop, which is useful for compiler developers.

The design of the system is centered around the idea that a MATLAB program is likely to be used on very different sized inputs, and hence at run-time loops will have very different loop bounds. Thus, our objective is to find important ranges for each loop nest, and to specialize the code for those ranges. Knowing the ranges for each specialization also enables us to use very fast and simple dependence testers.

The important phases of McFLAT, as illustrated in *Figure 3.1*, are the *Instrumenter*, which injects the profiling code, the *Range Estimator* which decides which ranges are important, and the *Dependence Analyzer and Loop Transformer Engine*. In the next section we look at each of these components in more detail.

3.3 Profiler

As illustrated in the phase labeled Instrument and Profile in *Figure 3.1*, the *Instrumenter* component is used to automatically inject instrumentation and profiling code into a MATLAB source file. This injection is done on the high-level structured IR produced by the McLAB front-end. In particular, we inject instrumentation to associate a unique loop number to each loop, and to gather following information for each loop.

- The lower bound of the iteration.
- The loop increment.
- The upper bound of the iteration.
- The nesting level of the loop.
- The time spent executing the loop.
- List of variables that are written to in the loop body. During the parallelism detection phase, this list of variables is required to ensure that within the list of indices for

> the arrays, exactly one index involves the loop variable and other variables used with loop index variable to index an array should remain constant over the entire execution of the loop.

The MATLAB program resulting from this instrumentation is functionally equivalent to the original code, but emits additional information that generates training data required for the next phase. *Listing 3.1* shows the instrumented source code file of our *Fiff* benchmark. When a source file is passed to the *Instrumenter* component of McFLAT, it traverses the AST(Abstract Syntax Tree) of the program. During traversal, when it encounters an AST node which is an instance of *ForStmt*, the instrumenter inserts a function call *xmlDataGenerator* after the loop body.

The inserted function call *xmlDataGenerator* is shown on lines 36 and 44. The first argument of the call is the name of .xml file which is generated and it has the same prefix as the source file. The second argument indicates the loop index variable name, in case of nested loop there are two loop index variables which are passed as a concatenated string using the MATLAB concatenation operator ([]). The third argument to the call is lower bound of the loop which in case of single loop is an integer value or the variable name but for a nested loop it is an array of integers created by concatenation operator. The fourth argument indicates the upper bound of the loop which is the variable name for single loops and an array of strings in case of a nested loop. The loop increment factor which is the fifth argument to *xmlDataGenerator* function is passed either as integer or an array of integers depending on the nesting level of the loop. The sixth argument indicates the loop number which uniquely identifies the loop in the program. The seventh argument indicates the maximum number of loops in the instrumented program. The eighth argument represents the list of variables that are written in the loop body. This is required for the parallelism detection phase. In the example from *Listing 3.1*, variables used to index the array are not rewritten within the loop body, thus the *VariablesUsed* argument to the call is empty. In cases where variables are written to within the loop body, the function call looks like as in *Listing 3.2*. In *Listing 3.2* *j* is assigned a value on line 6, so the last argument of the function call *xmlDataGenerator* is the name of the variable written in the loop body which in this case is *j*.

```
1
2  function U = finediff(a, b, c, n, m)
3  %-------------------------------------------------------------------------
4  %
5  % This function M-file finds the finite-difference solution
6  % to the wave equation
7  %
8  %   2
9  %  u (x, t) = c u (x, t),
10 %  tt xx
11 %
12 % with the boundary conditions
13 %
14 %  u(0, t) = 0, u(a, t) = 0 for all 0 ≤t ≤b,
15 %
16 %  u(x, 0) = sin(pi*x)+sin(2*pi*x), for all 0 < x < a,
17 %
18 %  u (x, 0) = 0 for all 0 < x < a.
19 %
20 %-------------------------------------------------------------------------
21
22 h = a/(n-1);
23 k = b/(m-1);
24 r = c*k/h;
25 r2 = r^2;
26 r22 = r^2/2;
27 s1 = 1-r^2;
28 s2 = 2-2*r^2;
29 U = zeros(n, m);
30 for i1 = 2:n-1,
31    U(i1, 1) = sin(pi*h*(i1-1))+sin(2*pi*h*(i1-1));
32    U(i1, 2) = s1*(sin(pi*h*(i1-1))+sin(2*pi*h*(i1-1)))+ ...
33    r22*(sin(pi*h*i1)+sin(2*pi*h*i1)+ ...
34    sin(pi*h*(i1-2))+sin(2*pi*h*(i1-2)));
35 end;
36 xmlDataGenerator('finediff', 'i1', 2, (n - 1), 1, 0, 1, 2,' ');
```

```
37
38 for i1 = 2:n-1,
39   for j1 = 3:m,
40   U(i1, j1) = s2*U(i1, j1-1)+r2*(U(i1-1, j1-1)+ ...
41   U(i1+1, j1-1))-U(i1, j1-2);
42    end;
43 end;
44 xmlDataGenerator('finediff', ['i1', ':', 'j1'], ([2, 3]'), ([n, m]'),
     ([1, 1]'), 1, 2, 2,' ');
```

Listing 3.1 Instrumented Source File

```
1 for i1 = 2:n-1,
2    U(i1, 1) = sin(pi*h*(i1-1))+sin(2*pi*h*(i1-1));
3    U(i1, 2) = s1*(sin(pi*h*(i1-1))+sin(2*pi*h*(i1-1)))+ ...
4    r22*(sin(pi*h*i1)+sin(2*pi*h*i1)+ ...
5    sin(pi*h*(i1-2))+sin(2*pi*h*(i1-2)));
6    j=i1;
7 end;
8 xmlDataGenerator('finediff', 'i1', 2, (n - 1), 1, 0, 1, 2,'j');
```

Listing 3.2 Instrumented Source File with Variables Written in Loop Body

The *xmlDataGenerator* is a function that is called once the loop finishes its execution. Within this function various tags are created as shown in *Listing 3.3*, and loop data is written to the .xml file. The *xmlDataGenerator* is optimized for write operations. It writes once to the .xml file when the number of loops is equal to the maximum number of loops in the program.

When the instrumented program is executed using a MATLAB virtual machine, the profile information is written to an .xml file. This .xml file is persistent, and so multiple runs can be made, and each run will add new information to the .xml file. *Listing 3.3* indicates the structure and the profiled information generated after the MATLAB source code file injected with instrumentation code is run. The .xml file starts with *RunNo* tag that indicate the Date and TimeStamp of the run. *Listing 3.3* shows there are two loops in the program,

and one of which is a nested loop. The *LoopNo* tag indicates the loop number that uniquely identifies the loop and *NestingLevel* indicate the level of the nested loop. For a loop with no nesting there is just one *NestedLoop* tag that contains the tags *LowerBound*, *UpperBound* and *LoopIncrementFactor*. The *LowerBound* tag is comprised of the loop index variable name *VariableName* and the value of lower bound. Similarly, the *UpperBound* contains the value that loop's upper bound is assigned during the execution of the program. The tag *LoopIncrementFactor* contains the factor by which the loop is incremented or decremented. For the case, where we have a nested loop with one level of nesting, there will be two counts of the tag *NestedLoop*.

```
<?xml version="1.0" encoding="utf-8"?>
<AD>
  <RunNo TimeStamp="28-Jun-2010 19:08:10">
    <LoopNo LoopNumber="1" NestingLevel="0" VariablesUsed="">
      <NestedLoop Number="1.0">
        <LowerBound>
          <VariableName>i1</VariableName>
          <start>2</start>
        </LowerBound>
        <UpperBound>
          <start>22</start>
        </UpperBound>
        <LoopIncrementFactor>
          <start>1</start>
        </LoopIncrementFactor>
      </NestedLoop>
    </LoopNo>
    <LoopNo LoopNumber="2" NestingLevel="1" VariablesUsed="">
      <NestedLoop Number="2.0">
        <LowerBound>
          <VariableName>j1</VariableName>
          <start>2</start>
        </LowerBound>
        <UpperBound>
```

```
            <start>23</start>
          </UpperBound>
          <LoopIncrementFactor>
            <start>1</start>
          </LoopIncrementFactor>
        </NestedLoop>
        <NestedLoop Number="2.1">
          <LowerBound>
            <VariableName>i1</VariableName>
            <start>2</start>
          </LowerBound>
          <UpperBound>
            <start>22</start>
          </UpperBound>
          <LoopIncrementFactor>
            <start>1</start>
          </LoopIncrementFactor>
        </NestedLoop>
      </LoopNo>
    </RunNo>
</AD>
```

Listing 3.3 Loop Profiling Information(.xml)

The loop profiling information .xml file is then used as an input to the next component which is the *Range Estimator*.

3.4 Range Estimator

The *Range Estimator* is the first important component of the main part of McFLAT, the Analysis and Transformations phase in *Figure 3.1*. The Range Estimator reads the loop profiling information and determines which are the important ranges for each loop. The important ranges are identified using Algorithm 1. The input to this algorithm is a hash table containing all the observed values for all the loops and the output is a list of important

ranges. The basic idea is that for each loop, we extract the observed values for that loop, partition the value space into regions and subregions, and then identify subregions which contain more values than a threshold.

3.4.1 Algorithm of Range Estimator

Algorithm 1 Algorithm for range estimation

```
Data Items
H (K,V) : Hash table with loop numbers as keys and list of observed values
Procedure processLoopData(LoopID)
l ← lookup(LoopID, H) // get all observed values for loop with LoopID
sort(l)
importantRanges ← empty
R ← computeRegions(min(l), max(l))
// for each large region
for all r in R do
   // for each subregion (divide R into 10 equal parts)
   for all sR in R do
      if numInRegion(l,sR) ≥ threshold then
         PredVal ← maxval(sR)
         add PredVal to importantRanges
      end if
   end for
end for
return(importantRanges)
```

We determine the regions and subregions as illustrated in *Figure 3.2*. The regions are powers of 10, starting with the largest power of 10 that is less than the smallest observed value, and ending with the smallest power of 10 that is greater than the highest observed value. For example, if the observed upper bounds were in the range 120 to 80000, then we would choose regions of size 100, 1000, 10000 and 100000. Each region is further subdivided into 10 subregions. A subregion is considered important if the number of observed values are above a threshold, which can be set by the user. For our experiments we used a threshold of 30 % . When an important region is identified, the the maximum observed value from the region is added to the list of important ranges.

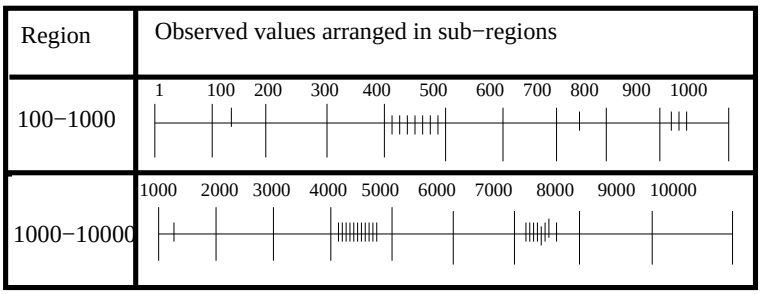

Figure 3.2 Pictorial Example of Ranges and Subranges

Listing 3.4 shows the format of .xml file which is generated as the Algorithm 1 is applied on the profiled information. The .xml file generated during this phase is persistent, and so multiple runs can be made, and each run will add new information under the *HD* tag of the .xml file. The .xml file starts with *RunNo* tag that indicates the Date and TimeStamp of the run. *Listing 3.4* shows the important loop-bound ranges for loops in the program, one of which is a single loop and the other one is a nested loop. There are two instances of the tag *LoopNo* for loop 1.0 which indicates that there are two important loop-bound ranges predicted for loop 1.0. The tag *LoopNo* has two elements *LoopNumber* that indicates the number that uniquely identifies the loop in the program and *LoopVariableName* which represents the name of the loop index variable. The tags *PredictedLowerBound*, *PredictedUpperBound* and *PredictedLoopIncFactor* within the *LoopNo* tag indicate important values observed for the lower bound, upper bound and loop increment factor of loop 1.0 respectively. Since loop 2 is a doubly-nested loop, so there are two *LoopNo* tags with different value for *LoopNumber* element. Which means there is one important value predicted for the outer loop 2.0 and one important value observed for the inner loop 2.1.

```
1 <HD>
2   <RunNo TimeStamp="2010-08-12 22:29:39">
3     <LoopNo LoopNumber="1.0" LoopVariableName="i1">
4       <PredictedLowerBound Value="2"/>
5       <PredictedLoopIncFactor Value="1"/>
6       <PredictedUpperBound Value="22"/>
7     </LoopNo>
```

```
8     <LoopNo LoopNumber="1.0" LoopVariableName="i1">
9       <PredictedLowerBound Value="2"/>
10      <PredictedLoopIncFactor Value="1"/>
11      <PredictedUpperBound Value="320"/>
12    </LoopNo>
13    <LoopNo LoopNumber="2.0" LoopVariableName="j1">
14       <PredictedLowerBound Value="2"/>
15      <PredictedLoopIncFactor Value="1"/>
16      <PredictedUpperBound Value="23"/>
17    </LoopNo>
18    <LoopNo LoopNumber="2.1" LoopVariableName="i1">
19       <PredictedLowerBound Value="2"/>
20      <PredictedLoopIncFactor Value="1"/>
21      <PredictedUpperBound Value="22"/>
22    </LoopNo>
23  </RunNo>
24 </HD>
```

Listing 3.4 Format of Predicted Loop-Bounds Ranges (.xml) File

3.5 Dependence Analysis

The data dependence testing problem is that of determining whether two references to the same array within a nest of loops may reference to the same element of that array [Wol90, ASU85].

Since we identify the upper loop bounds via our profiling phase, we have chosen very simple and efficient dependence testers: the *extended GCD test* and the *single variable per constraint test*. Currently, we have found these sufficient for our small set of benchmarks, but we can easily add further tests as needed. In proceeding sections, we will explain the theoretical aspects of above mentioned dependence tests with examples.

3.5.1 Extended GCD Test

We use Banerjee's Extended GCD Test [ASU85, Wol90] as a pre-processing step for our other tests. While the test itself is not exact, it allows us to transform our problem into a simpler and smaller form. This test solves the very simple question: Ignoring the bounds, is there an integral solution to a system of equations? If the system of equations is independent then the original system of equations is also independent. If this test returns dependence then the total system may be either dependent or independent. For example

```
1 for i=1:10
2  a(i+11)=a(i);
3 end
```

Listing 3.5 Extended GCD Example

The initial dependence problem is to find integers such i, i' such that $i+11=i'$ and
$1 \leq i,i' \leq 11$
The Extended GCD test tells us that $(i,i') = (t_1, t_1+11)$. Transforming the constraints to be in terms of t_1 gives us:
does there exist integer t_1 such that
$1 \leq t_1 \leq 11$
$1 \leq t_1+11 \leq 11$

This transformation is valuable for several reasons. First, the number of variables are reduced. Second, number of constraints have been reduced. Before this transformation, each lower and upper loop bound generated one constraint, while each dimension of the array generated one equality constraint. The equality constraint $a\vec{x} = b$ had to be converted into two inequality constraints $a\vec{x} \leq b$ and $a\vec{x} \geq b$. Therefore, there were $2*l+2*d$ (where l is the number of enclosing loops and d is the number of array dimensions) constraints. Now all the equality constraints are converted into bounds constraints. Thus there are only $2*l$ constraints left [MHL91].

3.5.2 SVPC: Single Variable Per Constraint Test

If the Extended GCD test returns dependence, then we apply SVPC test on each constraint comparing the upper and lower bounds. If after iterating over all the constraints, the lower bound(*lb*) is greater than the upper bound(*ub*) for any variable, then the test returns "independent". Otherwise the system of equations is dependent. This test also applies to many common multi-dimensional cases [MHL91]. To demonstrate the algorithm, we cover the following example in detail.

```
1
2 for i=1:10
3   for j=1:10
4     a(i,j) = a(i+10,j-9);
5   end
6 end
```

Listing 3.6 SVPC Example

The GCD test will set $i_1 = t_1, i'_1 = t_2, i_2 = t_2 + 9$ and $i'_2 = t_1 - 10$. Substituting the variables t_1 and t_2 into the above linear inequalities we get the following:

$1 \leq t_1 \leq 10$
$1 \leq t_2 \leq 10$
$1 \leq t_2 + 9 \leq 10$
$1 \leq t_1 - 10 \leq 10$

The first equation sets the lower bound of t_1 to 1 and the upper bound to 10. The second constraint also does the same for t_2. Thus combining the lower bounds from the last two inequalities with the upper bounds from the first two, we deduce
$10 \leq t_1 \leq 10$
$11 \leq t_2 \leq 10$

Since the lower bound on t_2 is greater than its upper bound, the system is independent. This algorithm is very efficient. It requires $O(num\_constraints + num\_vars)$ steps with many operations per step [MHL91].

During this phase of analysis and transformation, McFLAT, calculates dependencies between all the statements in the loop body against all the predicted important ranges for that loop using above mentioned dependence tests. It maintains various data structures supporting dependence analysis. This information is used in subsequent loop transformation phase. The output of this phase is .xml file whose structure is shown in *Listing 3.7*. The main tag of .xml file of Dependence Summary is *AD* which contains *LoopNo* tag having elements *LoopNumber* and *NestingLevel*. The *LoopNo* tag contains sub-tags *Range* having elements *Start*, *End* indicating the predicted lower bound and upper bound of the loop, whether a loop is parallelizable or not indicated by element *Parallelizable* and last element *ValidTransformation* which indicates the type of loop transformation applicable on the loop. In *Listing 3.7* there is only one sub-tag *Range* for loop 1.0 as there is only one range predicted for this loop. The number of instances of *Range* tag is equivalent to the number of loop bound ranges predicted for the loop. Within the *Range* tag, there is a *LoopStmts* tag that has elements called *Access* that represents the actual loop statement for which dependence information is calculated, and a *StmtNumbers* tag, that indicates the statements between which the dependence is calculated. In *Listing 3.7*, there are two statements within the loop body, so there are four *LoopStmts* tags which indicate the dependency between read(r) and write(w) of all statements in the loop body. In *Listing 3.7* the value of *StmtNumbers* element is "S1:S1" which shows the dependence information between r - w of the same statement which is S1. Similarly, the value of element *StmtNumbers* "S1:S2" shows dependence information between write of Statement1 and read of Statement 2. Likewise, the value of element *StmtNumbers* "S1:S2" shows the dependence information between write of Statement1 and write of Statement 2. The *Dependence* element indicates whether there is "dependence" or no "dependence" between the accessed statements. In case of loop number 2 which is a nested loop, a sub-tag *NestedLoop* is added to the *Range* tag to represent the dependence information for the nested loop.

```
1
2  <AD>
3   <LoopNo LoopNumber="1.0" NestingLevel="0">
4    <Range Start="2" End="22" Parallelizable="Yes"
        ValidTransformation="LoopReversal">
5      <LoopStmts Access="U(i1, 1)=(sin(((pi * h) * (i1 - 1))) + sin((((3
          * pi) * h) * (i1 - 1))))" StmtNumbers="S1:S1" Dependence="n" />
6      <LoopStmts Access="U(i1, 1) = sin(pi*h*(i1-1))+sin(2*pi*h*(i1-1));"
          StmtNumbers="S1:S2" Dependence="n"/>
7      <LoopStmts Access="U(i1, 1) = V(i1,1)" StmtNumbers="S1:S2"
          Dependence="n"/>
8      <LoopStmts Access="V(i1, 1) = sin(pi*h*(i1-1))+sin(2*pi*h*(i1-1));"
          StmtNumbers="S2:S2" Dependence="n"/>
9    </Range>
10 </LoopNo>
11  <LoopNo LoopNumber="2.0" NestingLevel="1">
12   <Range Start="2" End="23" Parallelizable="No"
        ValidTransformation="LoopInterchange">
13     <NestedLoop Number="2.1">
14       <Range Start="2" End="22.0" Parallelizable="Yes">
15         <LoopStmts Access="Vb(i1) =((U((i1 - 1), (j1 - 1)) + U((i1 + 1),
              (j1 - 1))) + (s2 * U(i1, (j1 - 1))))" StmtNumbers="S1:S1"
              Dependence="n"/>
16       </Range>
17     </NestedLoop>
18    </Range>
19   </LoopNo>
20 </AD>
```

Listing 3.7 Format of Dependence Summary (.xml) File

3.6 Loop Transformations

In our framework programmers can either suggest the type of transformation that they need to apply through optional loop annotations, or it will automatically determine and apply a

transformation or a combination of transformations which are legal for a loop.

McFLAT implements the following loop transformations that have been shown to be useful for two important goals parallelism and efficient use of memory hierarchy [LW04]: *loop interchange* and *loop reversal*. For automatic detection and application of the above mentioned loop transformations we use the unimodular transformation model presented in [WL91].

3.6.1 Distance Vector

Currently, McFLAT handles those loops whose dependences can be summarized by distance vectors. A *dependence distance* for a data dependence relation can be computed by finding the vector difference between the iteration vectors of the source and target iterations. The dependence distance will itself be a vector **d**, called the *distance vector*, defined as $\mathbf{d} = \mathbf{i^T} - \mathbf{i^S}$

$$\mathbf{i^S} + \mathbf{d} = \mathbf{i^T}$$

where $\mathbf{i^S}$ is the source iteration vector for the dependence relation, $\mathbf{i^T}$ is the target iteration vector.

Listing 3.8 shows an example of distance vector calculation. Are there iteration vectors i1 and i2, such that $2 \leq i_1 \leq i_2 \leq 4$ and $i_1 = i_2 - 1$?. The distance vector is $i_2 - i_1 = 1$

```
1 for i = 2:4
2   a(i) = b(i) + c(i);
3   d(i) = a(i-1);
4 end
```

Listing 3.8 Distance Vector Example

3.6.2 Loop Reversal

Loop reversal reverses the order in which values are assigned to the index variable of a loop. This subtle optimization can pave way for other optimizations and can eliminate dependencies. Our experimental results *Chapter 4*, have shown that this transformation can improve the performance of a MATLAB program, when applied to `for` loops that execute over a fairly large upper bound.

Figure 3.3 shows a code snippet with *Loop Reversal* applied on it. The original loop runs from 1 to 10 whereas *Loop Reversal* reverses the order in which values are assigned to the index variable, and the reversed loop runs down from 10 to 1.

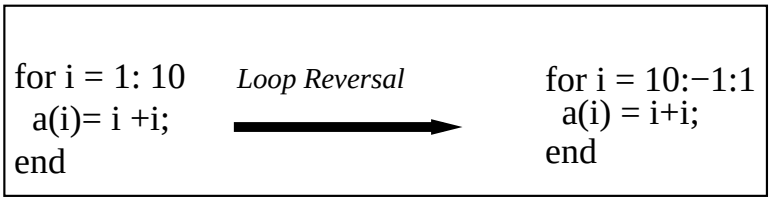

Figure 3.3 Example of Loop Reversal.

McFLAT, uses a unimodular transformations model for the application of *Loop Reversal*. In the context of unimodular transformations framework, loop reversal is represented by an identity matrix. Reversal of i^{th} loop is represented by the identity matrix, but with the i^{th} diagonal element equal to -1 rather than 1. For example, the matrix representing loop reversal of the outermost loop of a two-deep loop nest is $\begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}$.

3.6.3 Loop Interchange

Loop Interchange exchanges inner loops with outer loops. This transformation can improve the locality of reference, depending on how arrays are stored i.e. *column-major order* or *row-major order* in the programming language. This transformation is also known as *loop permutation*.

Figure 3.4 shows a code snippet with *Loop Interchange* applied on it.

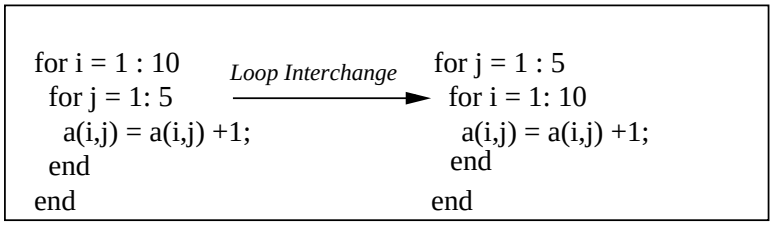

Figure 3.4 Example of Loop Interchange.

A loop interchange transformation maps iteration (i,j) to iteration(j,i). In matrix notation, we can write this as $\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} = \begin{bmatrix} j \\ i \end{bmatrix}$

The elementary transformation matrix thus performs the loop interchange transformation on the iteration space.

3.6.4 Legality of Unimodular Transformations

It has been proved that a loop transformation or a combination of loop transformations is *legal* if the transformed dependence vectors are all lexicographically positive. McFLAT uses the same legality test to determine whether a transformation or a group of transformation is valid for a loop or not [WL91].

Theorem* 1** *: Let* ***D **be the set of distance vectors of a loop nest. A unimodular transformation** ***T*** **is legal if and only if** $\forall d \varepsilon D : Td \succ 0$

Using the above theorem 1, we can evaluate if a compound transformation is legal directly or not. Consider the following example:

```
1
2 for i=1:N
3   for j=1:N
4     a(i,j) = a(i,j) + a(i+1,j-1);
5   end
6 end
```

Listing 3.9 Legality Test Example

The dependence vector of above code snippet is (1, -1). The loop interchange transformation, represented by $\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$, is illegal, since the resulting dependence vector $T(1,-1) = (-1,1)$ is lexicographically negative. However, applying loop interchange followed by reversal, represented by the transformation matrix

$T' = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}$

is legal since it leaves the resulting dependences $T(1,-1) = (1,1)$ lexicographically positive.

3.7 McFLAT: As a Test-Bed for Loop Transformations Application

Apart from automatically testing the legality of loop interchange and reversal or their combination, our framework supports a larger set of transformations which can be specified by the user. This allows us to use our system as a testbed for programmers with which they can suggest different transformations and observe the effect of different transformations on different loops. Programmers just have to annotate the loop body with the type of transformation that they need to apply on the loop. In *Listing 3.10* programmer asks the framework to apply *Loop Reversal* by annotating the loop body as mentioned in line 2. Our framework checks for the presence of annotations, if a loop annotation is present it computes the dependence information using the predicted loop bounds for that loop and applies the transformations if there is no dependency between the loop statements. The current set of transformations supported by annotations is:

- Loop fission: This transformation attempts to break a loop into multiple loops over the same index range. The fissioned loops will take only a part of the loop's body.

This transformation can reduce *cache misses*, as the most relevant data would be present in the cache for the split loops.

Figure 3.5 shows a code snippet with *Loop Fission* applied on it. This transformation breaks a loop into two loops, and each executing one statement of the previous loop which had two statements.

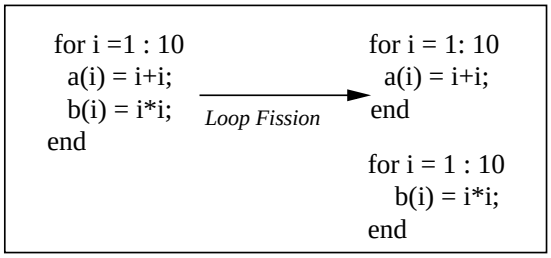

Figure 3.5 Example of Loop Fission.

- Loop fusion: A type of loop transformation that attempts to reduce code size and loop overhead by coalescing bodies of two loops into one. When two adjacent loops iterate the same number of times, their bodies can be fused as long as there is no dependency between the data items that these loops access.

  Figure 3.6 shows a code snippet with *Loop Fusion* applied on it. This transformation fuses two loops which iterate the same number of times into one loop.

- Loop interchange.

- Loop reversal.

```
1 for i1 = 2:(n-1),
2    (*LoopReversal;*)
3    U(i1, 1) = sin(pi*h*(i1-1))+sin(3*pi*h*(i1-1));
4 end;
```

Listing 3.10 A MATLAB loop with Annotated Loop Body

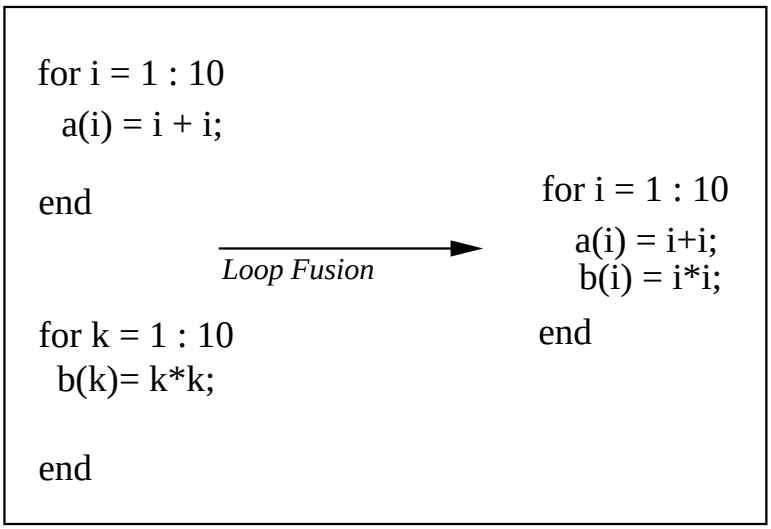

Figure 3.6 Example of Loop Fusion.

The above mentioned set of loop transformations are chosen because they have proved to improve the locality of reference and uncover parallelism opportunities[LW04]. The framework is extensible and any loop transformation can be implemented and their effects on different loop-bounds ranges can be studied.

McFLAT, is a useful tool that attempts to study the impact of different loop transformations on different loop-bound ranges. This knowledge is helpful for the application of loop transformations, as all legal transformations are not always beneficial for a loop. Our ultimate goal is to use the information provided by McFLAT in developing a self-learning system that will select optimal loop transformations based on the program features and loop-bounds that have been beneficial in the past for a transformation or a combination of transformations.

3.8 Output of McFLAT

McFLAT is a source-to-source framework which uses the McLAB front-end and outputs MATLAB code with loop-level specializations added for the important predicted ranges for that particular loop. *Listing 3.11* shows a code snippet from our *Crnich* benchmark. *Listing*

3.11 shows a single loop with no loop transformation applied.

```
1  function U = crnich(a, b, c, n, m)
2
3  h = a/(n-1);
4  k = b/(m-1);
5  r = c^2*k/h^2;
6  s1 = 2+2/r;
7  s2 = 2/r-2;
8  U = zeros(n, m);
9  t1=clock;
10 for i1 = 2:(n-1),
11    U(i1, 1) = sin(pi*h*(i1-1))+sin(3*pi*h*(i1-1));
12 end;
13 t2=clock;
14 fprintf(1, 'Time spent in loop: total = %f\n', (t2-t1)*[0 0 86400 3600
       60 1]');
15 end
```

Listing 3.11 Original Code

Listing 3.12 shows a code snippet from the *Crnich* benchmark, after it has passed through McFLAT. In the *Listing 3.12* the loop is guarded by a conditional check (line 9), that tests during the execution of the program that the value that is dynamically assigned to the upper bound variable *n* is equivalent to an already important predicted value, which in the example is 220. If the condition is satisfied, then a specialized version with loop transformation applied, which in this case is *Loop Reversal* is executed (lines 10 – 12). If the condition is not met then original loop with no loop transformation applied is executed (lines 13 – 16).

```
1  function [U] = crnich(a, b, c, n, m)
2    h = (a / (n - 1));
3    k = (b / (m - 1));
4    r = (((c ^ 2) * k) / (h ^ 2));
```

```
5   s1 = (2 + (2 / r));
6   s2 = ((2 / r) - 2);
7   U = zeros(n, m);
8   t1 = clock;
9   if (n ≤220)
10    for i1 = ((n - 1) : -1 : 2)
11      U(i1, 1) = (sin(((pi * h) * (i1 - 1))) + sin((((3 * pi) * h) * (i1
            - 1))));
12    end
13  else
14    for i1 = (2 : (n - 1))
15      U(i1, 1) = (sin(((pi * h) * (i1 - 1))) + sin((((3 * pi) * h) * (i1
            - 1))));
16    end
17  end
18 t2=clock;
19 fprintf(1, 'Total Time spent = %f\n', (t2 - t1)*[0 0 86400 3600 60 1]');
20 end
```

Listing 3.12 Specialized Code

3.9 Parallelism Detection

Compiler-based auto-parallelization is an area which has still not found wide-spread application [TWFO09]. This is due to the poor exploitation of application parallelism, which results in performance levels far below those obtained when original code is parallelized manually.

Recently, GPGPU and multi-core computing systems are widely seen as the most promising means of delivering performance with increasing transistor densities [Hof04]. However, this potential cannot be realized unless the application is well-parallelized, and the parallel constructs provided by the language takes advantage of the under-lying architecture efficiently.

Unfortunately, efficient automatic parallelization of a sequential program is a daunting and

complex task which needs to be done judiciously [TWFO09]. It is generally, agreed that manual code parallelization by expert programmers gives better performance and more streamlined parallel implementation, but at the same time the approach is most costly and time-consuming. Parallelizing compiler technology, on the other hand, can greatly reduce cost and time-to-market while ensuring semantic correctness of the resulting parallel code.

Automatic parallelism detection is a much studied research area [Lam74]. Progress was achieved in early 1980s and 1990s on restricted *DOALL* and *DOACROSS* loops [BC04, LL97]. In fact, this research has resulted in a whole range of parallelizing compilers, e.g. Polaris [PEH+93], SUIF [HAA+96] and Open64 [AL].

MATLAB is a popular programming language for numerical applications. Recently, compilers have been designed for MATLAB that attempt to improve its speed of execution and aim at exploiting parallelism opportunities either automatically or interactively [AP01, CB98, CB09].

3.9.1 Types of Dependency

Parallel computing is a form of computation that lets you do several calculations simultaneously, operating on the principle that large problems can often be divided into smaller tasks, which are then handled concurrently ("in parallel") [Par90]. There are several different forms of parallel computing: *bit-level*, *instruction-level*, *data*, and *task parallelism*.

Parallel computer programs are more difficult to write than sequential ones [PH05], because data dependencies in the programs introduce several new classes of potential software bugs. Communication and synchronization between the different smaller tasks are typically one of the greatest obstacles to getting good parallel programs which give better performance as compared to their sequential counterparts.

Understanding data dependencies is fundamental in applying loop transformations and writing parallel programs. There are four types of data dependencies.

- Flow Dependence: Also known as true dependence, statement *i* precedes statement *j*, and *i* computes a value that *j* uses. In *Listing 3.13* there is a flow dependence or true

dependence represented as $2 \rightarrow^{t} 3$. The dependence flows between instances of the statements in different iterations. This is a loop-carried dependence. The dependence distance is 1.

```
1 for i = 2:4
2   a[i] = b[i] + c[i];
3   d[i] = a[i-1];
4 end
```

Listing 3.13 Flow Dependency Example

- Anti Dependence: Statement i precedes j, and i uses a value that j computes. In *Listing 3.14* there is an anti-dependence, $2 \rightarrow^{a} 2$. This is a loop-carried dependence. The dependence distance is 1.

```
1 for i = 2:4
2  a[i] = b[i] + c[i];
3  d[i] = a[i+1];
4 end
```

Listing 3.14 Anti Dependency Example

- Input Dependence: Statement i precedes j, and i uses a value that j also uses. In *Listing 3.15* there is an input dependence between statements 3 and 4 denoted by $3 \rightarrow^{i} 4$.

```
1 x = 1;
2 y = x + 2;
3 x = z - w;
4 x = y / z;
```

Listing 3.15 Input Dependency Example

- Output Dependence: Statement i precedes j, and i computes a value that j also computes. In *Listing 3.16*, there is an output dependency between statements represented as $1 \rightarrow^o 3$ and $3 \rightarrow^o 4$.

```
1 x = 1;
2 y = x + 2;
3 x = z - w;
4 x = y / z;
```

Listing 3.16 Output Dependency Example

We refer to these dependency types in the *Experimental Results* chapter, where we list the type of dependencies present between loop statements of our benchmarks. McFLAT tests for the presence of flow dependency to determine whether a loop can be converted to a *parfor* loop or not.

McFLAT, which is the topic of this thesis is a source-to-source framework which uses McLAB front-end and outputs MATLAB code with loop-level specializations and automatically detects whether a loop can be converted to a **parfor** loop or not. The framework performs a parallelization test on the loops based on the dependence information calculated in the dependence analysis and instrumentation phase. *Listing 3.17* shows the syntax of the parallel for-loop represented by the keyword *parfor* in MATLAB. *parfor* has the same syntax for the range expression as the sequential *for* loop except the keyword *parfor* is used instead of *for* as shown on line number 1.

```
1 parfor i1 = 2:(n-1),
2    U(i1, 1) = sin(pi*h*(i1-1))+sin(3*pi*h*(i1-1));
3 end;
```

Listing 3.17 Syntax of Par-for loop in MATLAB

A loop is classified as a parallel loop in McFLAT, according to MATLAB's semantics [Mata], since the generated code is targeted for MATLAB system. Thus, a loop is clas-

sified as a parallel for-loop if it satisfies the following conditions.

- There should be no flow dependency between the same array access within the loop body. i.e. Distance vectors for all the same array accesses should be zero, e.g. `for` loop shown in *Listing 3.18* is not parallelizable due to the way variable *f* is used. This loop doesn't satisfy the flow dependency requirement of MATLAB's **parfor** loop.

```
for ii = 2:n
 for jj = 2:m,
  f(ii, jj) = f(ii, jj)+mask(ii, jj)* ...
  (0.25*(f(ii-1, jj)+f(ii+1, jj)+f(ii, jj-1)+ ...
  f(ii, jj+1))-f(ii, jj));
 end;
end
```

Listing 3.18 Example of Non-Parallelizable for Loop

The `for` loop shown in *Listing 3.19* satisfies the constraint of flow dependency for MATLAB's **parfor** loop.

```
for ii = 1:n,
   q = q+(f(ii, m)+f(ii+1, m))*0.5;
end;
```

Listing 3.19 Example of Parallelizable for Loop

- Within the list of indices for the variable, exactly one index involves the loop variable
- Other variables used with loop index variable to index an array should remain constant over the entire execution of the loop.
- The loop variable should not be modified in the body of the loop. This restriction is required, because changing loop variable in the parfor body invalidates the assumptions MATLAB makes about communication between the client and workers. The

`for` loop shown in *Listing 3.20* attempts to modify the value of the loop variable i in the body of the loop, and thus is invalid:

```
parfor i = 1:n
  i = i + 1;
  a(i) = i;
end
```

Listing 3.20 Example of Invalid parfor Loop

- The loop index variables must have consecutive increasing integers.

  parfor loopvar = initval:endval, statements, end

  allows you to write loops for a statement or block of code that executes in parallel on a cluster of workers, which are identified and reserved with the matlabpool command. initval and endval must evaluate to finite integer values, or the range must evaluate to a value that can be obtained by such an expression, that is, an ascending row vector of consecutive integers. The `parfor` loop shown in *Listing 3.21* fails the parfor range check.

```

parfor i = n:-1:1
  a(i) = i;
end
```

Listing 3.21 Example of Invalid parfor Loop

3.10 Limitations of McFLAT

At present, our framework implements a limited set of loop transformations. It only handles perfectly nested loops which have affine accesses and whose dependences can be summarized by distance vectors. As we develop the framework we will add further dependence

tests and transformations, as well as transformations to enable more parallelization. However, since we also wish to put this framework into our JIT compiler, we must be careful not to include overly expensive analyses.

Chapter 4

Experimental Results

In this chapter we demonstrate the use of McFLAT through two exploratory performance studies on a set of MATLAB benchmarks. We begin with a description of our benchmarks, their source, benchmark size, the versions of software used in performance measurements. We then present the types of dependence each loop carries within its statements and the number of statements within the loop body. This is followed by a comparison of performance numbers and speedups on Mathworks MATLAB, our McVM, McJIT and GNU Octave of transformed programs, applying our dependence testers and standard loop transformations for a variety of input ranges. The second study looks at the performance of benchmarks when we introduce **parfor** constructs on transformed loops and original loops. We then examine the factors that explain the performance numbers on different execution engines.

Our ultimate goal is to integrate McFLAT with a self-learning system that decides an optimal transformation for a loop based on its features, loop-bounds and system's past experience. However these example studies provide some interesting data and insight into how different loop-bound ranges, loop features impact various loop transformations. These exploratory studies provide an evidence that always applying loop transformations is not beneficial. At times loop transformations give no performance speedups and at times degrades the performance of the program.

4.1 Benchmarks and Static Information

Table 4.1 summarizes our collection of 10 benchmarks, taken from the McLab and University of Stuttgart benchmark suites. These benchmarks have a modest size, but yet perform interesting calculations and demonstrate some interesting behaviors. For each benchmark we give the name, description, source of the benchmark. *Table 4.2* lists the number of functions, number of loop nests and number of loops that can be automatically converted to parallel for loops.

We have chosen our benchmarks such that we can study the effect of loop transformations on the execution time of a program where loop body is representative of different features. In some of our benchmark loops computation is done on the same array within the loop body. Whereas for others there are no array dependencies. We have also included a benchmark in our suite which has loop that invokes a function and writes its return value to an array.

In *Table 4.3*, we report on the type of dependences that we observed within the statements in the loop body. The column labeled *Dep Type* indicates the type of dependences that exist within the loop body. In our benchmark suite the maximum number of loops that we have is five. The columns *Loop1*, *Loop2*, *Loop3*, *Loop4* indicate the number of statements in the loop body, for example, our benchmark *Crni* has four loops and the columns *Loop1*, *Loop2*, *Loop3*, *Loop4* indicate the number of statements in the loop body of each loop in *Crni* benchmark.

4.2 Performance Study for Standard Loop Transformations

For our initial study, we ran the benchmarks on an AMD Athlon™ 64 X2 Dual Core Processor 3800+, 4GB RAM computer running the Linux operating system; GNU Octave, version 3.2.4; MATLAB, version 7.9.0.529 (R2009b) and McVM/McJIT, version 0.5.

Benchmark Name	Source of Benchmark	Benchmark Description
Crni	McLab Benchmarks	Finds the Crank-Nicholson Sol.
Mbrt	McLab Benchmarks	Generates the mandelbrot set.
Fiff	McLab Benchmarks	Finds the finite-difference solution to the wave equation.
Hnormal	McLab Benchmarks	Normalises array of homogeneous coordinates.
Nb1d	McLab Benchmarks	Simulates the gravitational movement of a set of objects.
Interpol	Uni of Stutt	Compares the stability and complexity of Lagrange interpolation.
Lagrcheb	Uni of Stutt	Computes Lagrangian and Chebyshev polynomial for comparison.
Fourier	Uni of Stutt	Compute the Fourier transform with the trapezoidal integration rule.
Linear	Uni of Stutt	Computes the linear iterator.
EigenValue	Uni of Stutt	Computes the eigenvalues of the transition matrix.

Table 4.1 Description and Source of Benchmarks

For each benchmark we ran a number of training runs through the instrumenter and profiler and then we used our range estimator to predict ranges for the profile data. The Dependence analyzer and loop transformer use these ranges to generate a set of output files, one output file for each combination of possible transformations. For example, if the file had two loops, and loop reversal could be applied to both loops, then we would produce four different output files corresponding to: (1) no reversals, (2) reversing only loop 1, (3) reversing only loop 2, and (4) reversing both loops.

Each output file has a specialized section for each predicted important range, plus a dynamic guard around each specialized section to ensure that the correct version is run for a given input.

Benchmark Name	# Lines Code	# Funcs	# Loops	# Par Loops.
Crni	65	2	4	1
Mbrt	26	2	1	0
Fiff	40	1	2	0
Hnormal	30	1	1	1
Nb1d	73	1	1	0
Interpol	187	5	5	0
Lagrcheb	70	1	2	2
Fourier	81	3	3	2
Linear	56	1	2	1
EigenValue	50	2	1	0

Table 4.2 Characteristics of Benchmarks

We report the results for four different MATLAB execution engines, the Mathworks' MATLAB (which contains a JIT) (*Table 4.4*), the GNU Octave interpreter (*Table 4.5*), the McVM interpreter(*Table 4.7*), and the McVM JIT(McJIT) (*Table 4.6*). Execution time of benchmarks were averaged on three runs for all the four execution engines.

In each table, the column labeled *Trans. Applied* indicates which transformations are applied to the loops in the benchmark, where *N* indicates that no transformation is applied, *R* indicates Loop Reversal is applied, *F* represents Loop fusion and *I* is representative of Loop Interchange. *NN* indicates that there are two loops in the benchmark and no transformation is applied on any of them. Similarly, *IR* shows there are two loops, Interchange is applied on the first loop and reversal on the second loop. *I+R* indicates one loop nest on which interchange is applied and then reversal.

Benchmark	# Stmts in Loop Body							
Name	Loop1	Dep Type	Loop2	Dep Type	Loop3	Dep Type	Loop4	Dep Type
Crni	1	Nil	3	Nil	3	anti flow	1	anti
Mbrt	1	Nil						
Fiff	2	Nil	1	anti flow				
Hnormal	1	Nil						
Nb1d	18	output input						
Interpol	5	Nil	2	anti	2	anti	1	Nil
Lagrcheb	1	Nil	1	Nil				
Fourier	1	Nil	1	Nil	1	Nil		
Linear	1	flow	1	Nil				
EigenValue	1	Nil						

Table 4.3 Characteristics of Loops in Benchmarks

Depending on the benchmark we had two or three different ranges that were identified by the range predictor. The ranges appear in the tables in increasing value, so *Pred. Range 1* corresponds to the smallest range and *Pred Range 3* corresponds to the largest range. We chose one input for each identified range and timed it for each loop transformation version. In each table we give the speedup (positive) or slowdown (negative) achieved as compared to the version with no transformations. We indicate in bold the version that gave the best performance for each range.

Let us consider first the execution time for Mathworks' MATLAB, as given in *Table 4.4*. Somewhat surprisingly to us, it turns out that loop reversal alone always gives performance speed-up on the higher ranges. Whereas, on lower ranges there is either no speed up or performance de-gradation in some of the benchmarks. This implies that it may be worth having a specialized version of the loops, with important loops reversed for larger data ranges.

However, reversing one of the two loops having the same bounds and operating on the

Benchmark Name	Trans Applied	Pred. Range 1		Pred. Range 2		Pred. Range 3	
		Time	% Speedup	Time	% Speedup	Time	% Speedup
Crni	N	**60ms**		3.41s			
	R	60ms	0.0 %	**3.21s**	**5.8** %		
Mbrt	N	**1.91s**		9.40s			
	I	1.98s	-3.6 %	9.55s	-1.6%		
	R	1.91s	0.0 %	**9.25s**	**1.5**%		
	(I+R)	1.97s	-3.4%	9.32s	0.8%		
Fiff	NN	**400ms**		880ms			
	RN	405ms	-1.25%	**830ms**	**5.6**%		
Hnormal	N	1.85s		4.52s			
	R	**1.84s**	**0.5**%	**4.48s**	**0.8**%		
Nb1d	N	40ms		2.53s			
Interpol	N	44.70s		60.35s			
Lagrcheb	NN	140ms		280ms		450ms	
	RR	**138ms**	**1.4**%	**270ms**	**3.5**%	**420ms**	**6.6**%
	RN	143ms	-2.1%	280ms	0.0%	450ms	0.0%
	NR	143ms	-2.1%	280ms	0.0%	430ms	4.4%
Fourier	NNN	50ms		1.31s			
	FN	**40ms**	**20.0**%	1.49s	-13.7%		
	RRN	50ms	0.0%	1.25s	4.5%		
	(F+R)N	60ms	-20.0%	1.31s	0.0%		
	RNN	50ms	0.0%	**1.21s**	**7.6**%		
	NRN	50ms	0.0%	1.25s	4.5%		
Linear	NN	336ms		640ms		2.60s	
	IN	566ms	-68.4%	890ms	-39.0%	3.67s	-38.4%
	IR	610ms	-81.5%	850ms	-32.8%	3.42s	-31.5%
EigenValue	N	**80ms**		310ms		1.10s	
	I	100ms	-25.0%	370ms	-19.3%	1.18s	-7.27%
	R	90ms	-12.5%	290ms	6.4%	1.10s	0.0%
	(I+R)	90ms	-12.5%	**280ms**	**9.6**%	**1.08s**	**1.81**%

Table 4.4 Mathworks' MATLAB Execution Times and Speedups

Benchmark Name	Trans Applied	Pred. Range 1		Pred. Range 2		Pred. Range 3	
		Time	% Speedup	Time	% Speedup	Time	% Speedup
Crni	N	**5.46s**		1102s			
	R	5.46s	0 %	**1101s**	**0.09**%		
Mbrt	N	**289.8s**		**2000s**			
	I	300s	-3.5 %	2000s	0%		
	R	289.8s	0 %	2000s	0%		
	(I+R)	300s	-3.5%	2000s	0%		
Fiff	NN	6.44s		**251s**			
	RN	**6.41s**	**0.46**%	253s	-0.7%		
Hnormal	N	**7.34s**		**13.4s**			
	R	7.48s	-1.9%	13.6s	-1.4%		
Nb1d	N	2.56s		7.89s			
Interpol	N	3524s		5238s			
Lagrcheb	NN	**630ms**		1.28s		1.95s	
	RR	630ms	0%	**1.27s**	**0.7**%	**1.94s**	**0.51**%
	RN	630ms	0%	1.27s	0.7%	1.94s	0.51%
	NR	630ms	0%	1.27s	0.7%	1.94s	0.51%
Fourier	NNN	120ms		4.24s			
	FFN	120ms	0%	4.28s	-0.9%		
	RRN	120ms	0%	4.31s	-1.6%		
	FRN	120ms	0%	**4.19s**	**1.1**%		
	RNN	**110ms**	**8.3**%	4.26s	-0.4%		
	NRN	120ms	0%	4.25s	-0.2%		
Linear	NN	6.58s		**352s**		1496s	
	IN	6.65s	-1.0%	381s	-8.2%	1443s	3.5%
	IR	6.65s	-1.0%	382s	-8.5%	1422s	4.9%
	NR	**6.56s**	**0.3**%	369s	-4.8%	**1389s**	**7.1**%
EigenValue	N	240ms		**106s**		**460s**	
	I	**230ms**	**4.1**%	127s	-19.8%	502s	-9.1%
	R	230ms	4.1%	116s	-9.4%	486s	-5.6%
	(I+R)	230ms	4.1%	126s	-18.8%	507s	-10.2%

Table 4.5 Octave Execution Times and Speedups

		McVm(JIT)			
Benchmark Name	Trans Applied	Pred. Range 1		Pred. Range 2	
		Time	% Speedup	Time	% Speedup
Crni	N	**4.00s**		1074s	
	R	4.00s	0.0 %	**820s**	**23.6** %
Mbrt	N	**98.37s**		**675s**	
	I	101s	-3.3 %	714s	-5.8%
	R	110s	-12.6 %	781s	-15.6%
	(I+R)	106s	-8.16%	738s	-9.35%
Fiff	NN	**260ms**		500ms	
	RN	260ms	-1.95%	**460ms**	**8**%
Hnormal	N	5.00s		8.93s	
	R	**4.96s**	**0.8**%	**8.05s**	**10.9**%
Nb1d	N	850ms		4.10s	

Table 4.6 McVM(JIT) Execution Times and Speedups

		McVM(Interpreter)			
Benchmark Name	Trans Applied	Pred. Range 1		Pred. Range 2	
		Time	% Speedup	Time	% Speedup
Crni	N	7.12s		1386.2s	
	R	**6.35s**	**10.8** %	**1341.5s**	**3.2** %
Mbrt	N	384s		2491s	
	I	344s	10.4 %	**2286s**	**8.2**%
	R	**342s**	**10.9** %	2370s	4.8%
	(I+R)	346s	9.8%	2375s	4.6%
Fiff	NN	7.38s		7.46s	
	RN	**6.95s**	**5.8**%	**7.25s**	**2.8**%
Hnormal	N	7.23s		**11.6s**	
	R	**7.11s**	**1.6**%	12.24s	-5.5%
Nb1d	N	1.41s		4.24s	

Table 4.7 McVM(Interpreter) Execution Times and Speedups

same data item results in performance degradation at lower ranges, but has no impact as the bounds increase, for example, Lagrcheb, this is perhaps reversing one of the two loops results in more cache misses. In *Listing 4.1* benchmark, vector operations are performed within the loop body. A vector in MATLAB is defined as an array which has only one dimension with a size greater than one. For example, the array [1,2,3] counts as a vector. The following four vectors are worked on.

- *lag*
- *x*
- *x_plot*
- *y_plot*

for each value of j, it computes the difference $(d1)$ between the element $x\_plot(j)$ and *lag*; similarly, it computes the difference $(d2)$ between the element *x_i* and *lag*. It then computes an element-wise division by dividing $(d1)$ with the $(d2)$ to obtain a vector. The product of this vector is then assigned to the element $y\_plot(j)$.

```
1  function lebesgue = lagrcheb(n, i,scale)
2  % LAGRCHEB Compute and Plot Lagrangian and Chebyshev polynomial for
       comparison
3
4  lebesgue = 0;
5  n = fix(n);
6  i = fix(i);
7  if(n < 2), error('n<1'); end
8  if(i < 1 | i > n), error('i < 1 or i > n'); end
9
10 a = -1; % plot between a and b
11 b = 1;
12 m = 1e2; % default number of plot points between two nodes
13 clf;
14
```

```
15 x = linspace(a,b,n);
16 x_plot = linspace(a,b,m*n);
17 length(x_plot);
18 y_plot = zeros(1, length(x_plot));
19 lag = x([1:i-1 i+1:n]);
20 t1=clock;
21
22 for time=1:scale
23
24  for j=1:length(x_plot);
25    y_plot(j) = prod((x_plot(j)-lag)./(x(i)-lag));
26  end
27
28 x = cos((2*[n-1:-1:0]+1)./ (2*(n-1) + 2) * pi);
29 lag = x([1:i-1 i+1:n]);
30
31  for j=1:length(x_plot);
32    y_plot(j) = prod((x_plot(j)-lag)./(x(i)-lag));
33  end
34
35 end
36 t2=clock;
37 fprintf(1, 'total = %f\n', (t2-t1)*[0 0 86400 3600 60 1]');
```

Listing 4.1 Lagrcheb Benchmark

MATLAB accesses arrays in column-major order, and MATLAB programmers normally write their loops in that fashion, so always applying loop interchange degrades the performance of the program. Performance degrades more for loops which involve array dependencies. However, the degradation impact is lower at higher ranges perhaps due to cache misses in both the cases, that is transformed and original loop. The loop interchange degradation impact is less for loops that invoke a function whose value is written to an array, for example, Mbrt. In *Listing 4.2*, the loop on lines from $10-14$ is a candidate for loop interchange transformation. Within the loop body, a function named *iteration* is called and its return value is written to an array named *set*

```
function set=mandelbrot(N, Nmax)
  side = round(sqrt(N));
  ya = -1;
  yb = 1;
  xa = -1.5;
  xb = .5;
  dx = (xb-xa)/(side-1);
  dy = (yb-ya)/(side-1);
  set = zeros(side,side);
  for x=0:side-1
    for y=0:side-1
      set(y+1,x+1) = iterations(xa+x*dx+i*(ya+y*dy),Nmax);
    end
  end
end

function out = iterations(x,max)
  c = x;
  i = 0;
  while(abs(x) < 2 & i < max)
    x = x*x + c;
    i = i+1;
  end
  out = i;
end
```

Listing 4.2 Mbrt Benchmark

Loop fusion was only applied once (in Fourier) where it gives a performance speed-up on lower ranges. However, as the loop bounds and accessed array get bigger then performance degrades.

Now consider the execution time for Octave, given in *Table 4.5*. Octave is a pure interpreter and you will note that the absolute execution times are often an order of magnitude slower than Mathworks' system, which has a JIT accelerator. The applied transformations also seem to have very little impact on performance, particularly on the lower ranges. For

higher ranges, no fixed behavior is observed, for some benchmarks there is a performance improvement whereas for others performance degrades.

We were also interested in how the transformations would impact our group's McVM, both in pure interpreter mode, and with the JIT. We couldn't run all the benchmarks on McVM because the benchmarks use some library functions which are not currently supported. However, *Table 4.6* and *Table 4.7* lists the results on the subset of benchmarks currently supported. Once again loop reversal can make a significant impact on the larger ranges for the JIT, and actually also seems beneficial for the McVM(interpreter).

4.3 Performance study for Parallel For Loops

In *Table 4.8* we report the execution time and speedups with MATLAB's `parfor` looping construct. We ran the benchmarks on an Intel ™Core(TM) i7 Processor (4 cores), 5.8GB RAM computer running a Linux operating system; MATLAB, version 7.9.0.529 (R2009b). For these experiments we initialized the MATLAB worker pool to size 4.

The term pN indicates that there is one loop in the benchmark, which is parallelized and no loop transformation is applied on it. (pF) means two loops are fused and then the fused loop is parallelized. Note that it is not possible to combine loop reversal and parallelization with the MATLAB `parfor` construct as the MATLAB specifications require that the loop index expression must increase.

We have reported execution times of various combinations of parallel and sequential loops, to study the effect of parallelizing a loop in the context of MATLAB programming language.

For most of the benchmarks we observed that MATLAB's `parfor` loop does not often give significant performance benefits, and in some cases causes severe performance degradation. This is likely due to the parallel execution model supported by MATLAB which requires significant data copying to and from worker threads that overshadows the gain achieved by executing the tasks in parallel.

MATLAB's **parfor** loop has its advantages and disadvantages, so automatically converting

Benchmark Name	Trans Applied	Pred. Range 1		Pred. Range 2		Pred. Range 3	
		Time	% Speedup	Time	% Speedup	Time	% Speedup
Crni	N	**280ms**		13.41s			
	pN	1.03s	-257%	14.20s	-5.9%		
	R	290ms	-3.5 %	**13.30s**	**0.8** %		
Hnormal	N	800ms		1.70s			
	pN	70.5s	-8712 %	71.3s	-4094%		
	R	**780ms**	**2.5**%	**1.68s**	**1.1**%		
Lagrcheb	NN	120ms		200ms		280ms	
	(pN)(pN)	140ms	-16.6%	**180ms**	**10.0**%	**250ms**	**10.7**%
	N(pN)	**110ms**	**8.3**%	180ms	10.0%	250ms	10.7%
	(pN)N	120ms	0.0%	180ms	10.0%	260ms	7.1%
	R(pN)	120ms	0.0%	180ms	10.0%	250ms	10.7%
	(pN)R	120ms	0.0%	180ms	10.0%	250ms	10.7%
	RR	120ms	0.0%	200ms	0.0%	270ms	3.5%
	RN	130ms	-8.3%	200ms	0.0%	270ms	3.5%
	NR	130ms	-8.3%	200ms	0.0%	270ms	3.5%
Fourier	NNN	170ms		**680ms**			
	(pN)NN	50ms	70%	720ms	-5.8%		
	(pN)(pN)N	200ms	-17.6%	720ms	-5.8%		
	N(pN)N	50ms	70%	720s	-5.8%		
	(pF)N	50ms	70%	720ms	-5.8%		
	R(pN)N	50ms	70%	710ms	-4.4%		
	(pN)RN	50ms	70%	680ms	0.0%		
	FN	**20ms**	**88.2**%	690ms	-1.4%		
	RRN	170ms	0.0%	680ms	0.0%		
	(F+R)N	170ms	0.0%	680ms	0.0%		
	RNN	170ms	0.0%	680ms	0.0%		
	NRN	170ms	0.0%	680ms	0.0%		
Linear	NN	150ms		7.40s		29.8s	
	N(pN)	**150ms**	**0.0**%	**7.20s**	**2.7**%	30.2s	-1.3%
	I(pN)	390ms	0.0%	10.30s	-39.1%	40.2s	-34.8%
	IN	370ms	-146.6%	10.30s	-39.1%	37.6s	-26.1%
	IR	370ms	-146.6%	10.30s	-39.1%	37.6s	-26.1%
	NR	160ms	-6.6%	7.20s	2.7%	**29.4s**	**1.34**%

Table 4.8 Mathworks' MATLAB Execution Times and Speedups with Parallel Loops

a sequential loop into a **parfor** loop would not always be beneficial.

Chapter 5

Related Work

In this chapter, we discuss various compilers that have been designed to compile dynamic languages like MATLAB, to compile either ahead of time or just-in-time to avoid the overhead of interpreting. Then we discuss various dependence testing algorithms that are being used to compute dependence between same array references. Then we talk about various approaches that have been used previously to determine optimal loop transformations based on their impact on the program. McFLAT, introduces loop-level specializations for important predicted ranges in MATLAB programs, so we talk about program specialization techniques used previously for different languages to gain performance speedups. Then we discuss previous approaches used to detect automatically parallelization opportunities in programs. The last section discusses techniques used for adaptive compilation in the context of different programming languages.

Previous compilers have tried to gain performance speedups by translating MATLAB code to other static languages, such as C [JB07] or Fortran 90 [RGG+96, DRP96]. This approach allows other optimizations and parallelization that can be done on static languages, as more information is available about the translated program. Lately, code restructuring is performed for MATLAB programs to take advantage of language optimized operations e.g. Vectorization of loops [BLA07].

Falcon [RGG+96]: A MATLAB Interactive Restructuring Compiler, provides a program-

ming environment that uses an existing high-level array language MATLAB as a source language and generates Fortran 90 programs with directives for parallelism. Falcon performs static, dynamic and interactive analysis to generate the output code. It includes capabilities for interactive and automatic transformations at both the operational-level and the functional-level which results in better performance.

Menhir: An environment for high-performance MATLAB [CB98] . It is a compiler for generating sequential or parallel code from the MATLAB language. The compiler is designed such that it takes MATLAB as a specification language and generates parallel and sequential C or Fortran code.

McLAB, [mcl] an endeavor of Sable lab at McGill University aims to provide languages, compilers and virtual machine for dynamic scientific languages. Starting with the MATLAB language, and extensions of the MATLAB language such as Aspect Matlab [TAH10]. McLAB also has a compiler for generating Fortran 95 code called McFor [Li09]. The McLAB virtual machine (McVM) currently integrates an interpreter and an optimizing JIT compiler(McJIT) supporting a non-trivial subset of the MATLAB programming language. McFLAT, the topic of this thesis is part of analysis and transformation engine of McLAB having a dependence analyzer, a basic parallelization detection mechanism and a loop transformer component. McFLAT, uses profile-based training runs to collect information about loop bounds and ranges, and then applies a range estimator to estimate which ranges are most important. Specialized versions of the loops are then generated for each predicated range. Our ultimate goal is to embed this framework in our McJIT system, where it will work as an adaption system invoking the compiler to recompile a method for performance speedups based on what it has seen in the past.

5.1 Efficient And Exact Dependence Analysis

There is a rich body of research on the topics of dependence analysis, loop transformations and parallelization. In our related work, we attempt to cover a representative subset that, to the best of our knowledge, covers the prior work in the area of this thesis.

Data dependence testing is the first step in detecting loop level parallelism in numerical computation. "The problem is equivalent to integer linear programming and thus in general cannot be solved efficiently" [MHL91]. The most efficient known dependence testing algorithms either depend on the value of the loop bounds or are order $O(n^{o(n)})$ where n is the number of loop variables [Kan87, Len83, Sch86]. Many algorithms have been proposed for this (dependence testing) problem, each one selecting different tradeoffs between accuracy and efficiency. Traditional algorithms attempt to prove independence, but in case of failure they assume dependence [AK87, Ban88, Wal88, Wol90]. If such an algorithm returns dependent, we are not sure if an approximation was made or the set of constraints are actually dependent.

Some work has been done on algorithms which are guaranteed to be exact for special case inputs [Sho81]. However, [MHL91] uses a series of special case exact tests. If the input is not of the appropriate form for an algorithm, then they try the other next one. Using a series of tests allows them to be exact for a wider range of inputs. Cascading exact tests can also be much more efficient than cascading inexact ones. By attempting the most applicable and least expensive test first, in most cases they return a definitive answer using just one exact test. McFLAT, uses the same approach as in [MHL91], we have implemented a set of dependence tests, which we have tested on our benchmark suite and found the algorithms efficient and exact for our input set.

Several other well-known dependence tests are

- *ITest*: [PP94] is an optimized test which combines the GCD and the Banerjee tests. Whenever either of the GCD test or the Banerjee test produces a "no" answer, the ITest gives the same answer. In a number of cases where the GCD and the Banerjee tests produce a "maybe" answer the I test, produces a no. In addition it is able to produce a definite "yes" answer when the GCD and Banerjee tests produce only a "maybe".

- *Fourier−Motzkin Test* : solves the general non-integer linear programming case exactly. If the result of this test is independent, the integer case is also independent [DE73]. In case this test returns dependent, it also returns a sample solution. If this

sample solution is integral, then the integral case is dependent. Otherwise, this test is not exact.

- *Omega Test*: is based on Fourier-Motzkin variable elimination to integer programming. However, the Omega test is more promising as compared to Fourier-Motzkin test as it, combines new methods for eliminating equality constraints with an extension of Fourier-Motzkin variable elimination to integer programming [Pug91]. The Omega test determines whether there is an integer solution to a problem which in its case is a set of linear equalities and inequalities. The input to the Omega test is a set of linear equalities and inequalities.

5.2 Loop Transformations

Several techniques are used to decide the order of loop transformations. A technique commonly used in parallelizing compilers is to decide *a priori* the order in which the compiler should attempt to apply transformations. This technique is inadequate and inefficient because the choice and ordering of loop optimizations are highly dependent on program semantics, and the optimality of a transform cannot be evaluated locally, one step at a time [WL91].

Another used technique is to “generate and test”, that is, to exhaustively explore all different possible combinations of transformations. This “generate and test” approach is expensive and also cannot search the entire space of transformations that have potentially infinite instantiations. Another disadvantage of this approach is that differently transformed versions of the same program may trivially have the same behavior and need not be explored [WL91].

Linear transformations are widely used to vectorize and parallelize loops. A smaller set of these transformations are unimodular transformations. Unimodular loop transformations have been widely used since they reduce the problem of applying and testing the legality of loop transformations to matrix operations, thereby, allowing the application of many useful loop transformations efficiently[FLVG95].

Loop interchange, reversal and skewing transformations are modeled as unimodular transformations in the iteration space [FM85, Qui84, DS90, WL91]. A compound transformation is just another linear transformation, being a product of several elementary transformations. This model provides a general legality test for a single or a compound transformation, as opposed to a specific legality test for an individual elementary transformations. The loop transformer component of our McFLAT framework also uses a unimodular transformations model to apply and test the legality of a loop transformation or a combination of loop transformations, but our intent is to specialize for different predicted loop bounds.

5.3 Impact of Loop Transformations

Optimizations are applied at various levels to gain performance speedups.

High-order transformations are optimizations that specifically improve the performance of loops through techniques such as loop interchange, loop fusion, and loop unrolling. The goals of these loop optimizations include reducing the costs of memory access through the effective use of caches, overlapping computation and memory access through effective utilization of the data prefetching capabilities provided by the hardware. Improperly selected high-order transformations can degrade the performance to an extent worst than the same unoptimized code [Sar97]. Hence, automatic selection of high-order transformations has to be done judiciously to get the desired benefit.

Pre-processors have been designed that apply various transformations on the source code or intermediate representation of the source code, which when compiled by the native compiler generates better and optimized code. Kuck and Associates Pre-processor (KAP) is an optimizing pre-processor that applies various transformations on the source code (e.g. temporary variables induction). This pre-processor is an integral part of HP's Fortran compiling systems, and if used with proper pre-processing directives have given a performance improvement on computationally intensive tasks. VAST, also a pre-processor, operates through a compiler driver on either intermediate representations of the program (such as Edison Design Group IR) or directly on source code to perform high-level optimizations,

which includes loop nest optimizations and automatic vectorization and parallelization.

Optimizing compilers use either profiled information of previous runs or compiler analysis to estimate the execution time cost, memory cost, semantics of underlying architecture and execution frequencies. Individual program transformations are used in different ways to satisfy different optimization goals. Quantitative models based on memory cost analysis have also been used to select optimal loop transformations. Memory cost analysis chooses a beneficial transformation based on the number of distinct cache lines and the number of distinct pages accessed by the iterations of a loop [Sar97].

Locality optimization in the SUIF (Stanford University Intermediate Format) 1.x compiler performs unimodular + tiling transformations. Selection of unimodular transformation is based on identifying reusability of vector space [LW04]. Schreiber and Dongarra addressed problem of selecting optimal tile sizes to minimize communication traffic and cache misses. But their analysis is restricted to isomorphic iteration and data spaces [DS90].

Another framework presented in [ZCS03] predicts the impact of optimizations for some objective (e.g., performance, code size or energy). The framework consists of three types of models: optimization models, code models and resource models. By integrating these models, a benefit value is calculated that represents the benefit of applying an optimization in the context of given code for the objective represented by the resources.

Our framework, McFLAT, is a preliminary step towards building a self-learning system that selects optimal transformations based on loop bounds and profiled program features that have been beneficial in the past for a transformation or a combination of transformations.

5.4 Program Specializations

Procedure cloning is an interprocedural optimization technique by which a compiler can create specialized versions of function bodies called “clones”. Each clone expects different set of parameters on the entry to the procedure. This paves way for further optimizations of the procedure body. The call sites are then modified to call the appropriately optimized version of the procedure [CHK93].

Dynamic dispatching is a major performance bottle-neck for programs written in object-oriented style. The cost of performing method look-up makes dynamically dispatched calls (also known as virtual function calls and message sends) expensive. To avoid this cost, optimizing compilers for object-oriented languages try to statically-bind as many message sends as possible to called methods. Static binding requires class information, so that set of possible invocable methods can be determined and message sends can be bound statically. One way of improving the precision of class information, and indirectly to support more static binding, is to compile multiple specialized versions of a method, and each method operates on a different input arguments [DCG95]. Another program specialization technique called "customization" is also used to compile a specialized version of a method for each possible receiver class, and methods are never specialized for arguments other than the receiver [CU89].

[DCG95] have designed a goal-directed algorithm, that uses dynamic profile data to specialize hot methods rather than specializing exhaustively. Selective method specialization approach takes into account the cost and benefits of generating a specialized version of a method using profile data. Therefore, this technique does not suffer from serious code explosion and also does not generate identical multiple specialized versions which could be coalesced into one without a significant impact on program performance.

The semantics of dynamic programming languages make them a good candidate for a variety of optimizations including program specialization not only at the function level, but also at the loop level. In case of dynamic languages, the loop body will be interpreted and executed line-by-line which degrades the performance of programs written in these languages. To reduce the interpretive overhead various techniques like vectorization [BLA07] and use of profiled data to gather more information about the original program [AH10] have been employed.

McFLAT, generates multiple versions of the original source code which are specialized at the loop level. The main idea is to collect information about loop-bounds and then decide heuristically which loop-bound ranges are worth specializing using a variety of loop transformations. We observed significant speedups for the specialized versions, and noted that loop transformations had different impacts depending on the loop range and execution

engine. Our ultimate goal is to embed this framework in our McJIT system.

5.5 Automatic Parallelism Detection and Vectorization

Static automatic parallelism extraction has been achieved in the past[BC04, LL97]. Unfortunately, many parallelization opportunities could still not be discovered by a static analysis approach due to lack of information at the source code level. Tournavitis et. al. have used a profiling-based parallelism detection method that enhances static data dependence analysis with dynamic information, resulting in larger amounts of parallelism uncovered from sequential programs [TWFO09]. McFLAT, also uses profiling-based parallelism detection but in the context of MATLAB programming language and within the constraints of MATLAB parallel loops.

[KPW+07] argue that current implementations of optimistic techniques such as thread-level speculation cannot uncover all opportunities of parallelism because they do not use the proper abstractions for the data structures in the programs. Kulkarni et. al. [KPW+07] suggested an object-based optimistic parallelization system for irregular applications that manipulate pointer-based data structures. "The Galois system" is an object-based shared-memory model, which allows concurrent accesses and updates to shared objects by exploiting the high level semantics of abstract data types.

A dimension abstraction approach for vectorization in MATLAB presented in [BLA07] discovers whether dimensions of an expression will be legal if vectorization occurs. The dimensionality abstraction provides a representation of the shape of an expression if a loop containing the expression was vectorized. To improve vectorization in cases which have incompatible vectorized dimensionality, a loop pattern database is provided which is capable of resolving obstructing dimensionality disagreements.

5.6 Adaptive Compilation

Heuristics and statistical methods have already been used in determining compiler optimization sequences. For example, Cooper et. al. [CSS99] developed a technique using genetic algorithms to find "good" compiler optimization sequences for code size reduction.

Previous adaptive virtual machines have used method invocation counters to identify hotspots in the program and optimize them. Holzle and Unger [HU96] describe SELF-93 system, an adaptive optimization system for SELF language. The goal of the project is to avoid long pauses in interactive applications by optimizing only performance-critical parts of the application. Method invocation counters with an exponential decay mechanism are used to identify candidates for optimization.

Whaley [Wha00], implemented sample based calling-context-sensitive profiling in a production JIT compiler. They have empirically demonstrated that their profiling technique has low overhead and can give performance gain at startup and steady-state.

[Wha00, GDGC95] have explored off-line profile directed compilation techniques that use one or more profiles from previous runs of an application as a feedback into a compiler to make improved optimization decisions for future executions. Such systems include Digital FX ! 32 [HH97], Morph [ZWG+97], and DCPI [ABD+97]. Jalapeo JVM [AFG+00] uses an adaption system that can invoke a compiler when profiling data suggests that recompiling a method with additional optimization will be more beneficial. Our work is a first step towards developing an adaptive system that will be embedded in our McVM, that applies loop transformations based on predicted data from previous execution runs and profiled information about the programs.

Previously work has been done on JIT compilation for MATLAB. MaJIC [AP01], combines JIT-compilation with an offline code cache maintained through speculative compilation of Matlab code into C/Fortran. It derives the most benefit from optimizations such as array bounds check removals and register allocation. Mathworks introduced the MATLAB JIT-Accelerator [Mat02] in MATLAB 6.5 that has accelerated the execution of MATLAB code. McVM [CBHV10, CB09] is also an effort towards JIT compilation for MATLAB, it

uses function specializations based on run-time type of their arguments. McVM(JIT) has shown performance speed-ups against MATLAB for some of our benchmarks. McFLAT, the framework which is the topic of this thesis uses profiled program features and heuristically determines loop bounds ranges to generate specialized versions of loops in the program.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

In this chapter, we start with an overview of the driving principle in the designing of a framework for MATLAB loop analysis and transformations the McFLAT. We then discuss the contributions made by this thesis. Then, we briefly, discuss different phases of McFLAT. We conclude with a discussion of results on four different execution engines. In the future work section, we discuss possible enhancements that can be done in McFLAT.

Parallelization and optimization of the MATLAB programming language presents several challenges due to the dynamic nature of MATLAB. Since MATLAB does not have static type declarations, neither the shape and size of arrays, nor the loop bounds are known at compile-time. This means that many standard array dependence tests and associated transformations cannot be applied straight-forwardly. On the other hand, many MATLAB programs operate on arrays using loops and thus are ideal candidates for loop transformations and possibly loop vectorization/parallelization. McFLAT, was designed to gain performance speedups by applying loop transformations. The main hurdle in achieving this goal was the lack of information about loop bounds which are required for dependence testing.

We have described a new framework, McFLAT, which uses profile-based training runs to collect information about loop bounds and ranges, and then applies a range estimator to estimate which ranges are most important. Specialized versions of the loops are then generated for each predicated range. The MATLAB code generated from McFLAT can be run on any MATLAB virtual machine or interpreter.

Results obtained on four execution engines (Matlab, GNU Octave, McVM(JIT) and McVM(interpreter) suggest that the impact of different loop transformations on different loop bounds is different and also depends on the execution engine. We were somewhat surprised that loop reversal was fairly useful for several execution engines, especially on large ranges. The framework also detects whether a loop is parallelizable or not. It detected quite a few parallel loops and transformed them to MATLAB's `parfor` construct, the execution benefit was very limited and sometimes very detrimental. Thus our McJIT compiler will likely support a different parallel implementation which has lower overheads.

6.2 Future Work

In this section we look into possible improvements to different components of McFLAT. McFLAT has the potential to be further evaluated and its functionality can be enhanced which can make it an important component of our McVM.

Although McFLAT is already a useful stand-alone tool, in our overall plan it is a preliminary step towards developing a self-learning system that will be part of our McJIT. This adaptive system will decide on whether to apply a loop transformation or not depending on the benefits that the system has seen in the past and will suggest recompilation of code to gain performance speedups.

As for Instrumenter is concerned, it can be extended to extract more features about program loops, which can be useful for dependence analyzer and loop transformer. Information like whether loop body invokes a function or operates on arrays can be useful for loop transformer. This is because impact of different loop transformations varies according to the computation done in the loop body. When deciding on which ranges to specialize, it

could be beneficial to focus on ranges which are likely to exhibit different cache behavior.

Many of the integer linear programs from the data dependence consists of inequalities that involve only one unknown [ASU85]. The programs can be solved simply by testing if there are integers between the constant upper bounds and constant lower bounds independently. Currently, McFLAT implements a few dependence tests that have proved to be exact for our benchmark suite. Other dependence tests like the *Acyclic Test*, the *The Loop Residue Test* can be implemented fairly easily in our framework.

Currently, McFLAT supports a limited set of loop transformations. Other loop transformations like *Loop Skewing* and *Tiling* can be added to increase parallelism opportunities.

Our initial exploratory experiments validate that different loop transformations are beneficial for different ranges. Future work will focus on extracting more information about the program features from profiling, maintaining a mapping between loop bounds, program features and effective loop transformations and making use of past experience to make future decisions on whether to apply transformations or not.

Appendix A

User Manual

McFLAT, is executed using the “Main” entry point of the McLAB project. “-danalysis” flag indicates to run McFLAT.

A.1 Flags

McFLAT supports the following list of flags:

- `-m -danalysis -dir -prof crni`

  Above mentioned flag ”-prof” is used to instrument original MATLAB code which when executed ejects additional information. ”-dir” indicates that the starting point is a directory structure. If the ”-dir” flag is not there, then McFLAT expects a ”.m” file as an input. As a result of this phase a directory with the name *Dep⟨ BenchmarkName ⟩* is created.

- `-m -danalysis -dir -heur crni`
  ”-heur” flag indicates to run *Range Estimator* on the Loop profiling Information (.xml).

- `-m -danalysis -dir -auto crni`
  ”-auto” flag indicates to apply loop transformations automatically on the input ”.m” files.
- `-m -danalysis -dir -anno crni`
  ”-anno” flag indicates to apply only those loop transformations that are annotated in the loop body.

`languages/Natlab/src/natlab/toolkits/DependenceAnalysis` - McFLAT source Java files, which includes the complete McFLAT system.

Specialized versions of ”.m” file will be generated in *Dep*⟨ *BenchmarkName* ⟩ and they can be executed on any MATLAB systems.

Bibliography

[ABD+97] Jennifer M. Anderson, Lance M. Berc, Jeffrey Dean, Sanjay Ghemawat, Monika R. Henzinger, Shun-Tak A. Leung, Richard L. Sites, Mark T. Vandevoorde, Carl A. Waldspurger, and William E. Weihl. Continuous profiling: where have all the cycles gone? *ACM Trans. Comput. Syst.*, 15(4):357–390, 1997.

[AFG+00] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive optimization in the Jalapeno JVM. In *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, Minneapolis, Minnesota, United States, 2000, pages 47–65. ACM, New York, NY, USA.

[AH10] Amina Aslam and Laurie Hendren. McFLAT: A profile-based framework for matlab loop analysis and transformations. Technical Report SABLE-TR-2010-6, Sable Research Group, School of Computer Science, McGill University, Montréal, Québec, Canada, 2010.

[AK87] Randy Allen and Ken Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM Trans. Program. Lang. Syst.*, 9(4):491–542, 1987.

[AL] Computer Architecture and Parallel Systems Laboratory. Open64. Home page `http://www.open64.net`.

[AP01] George Almasi and David A. Padua. MaJIC: A MATLAB just-in-time compiler. In *Languages and Compilers for Parallel Computing*, 2001. Springer Berlin / Heidelberg.

[ASU85] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison Wesley, 1985.

[Ban88] Utpal K. Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, Norwell, MA, USA, 1988.

[BC04] Michael G. Burke and Ron K. Cytron. Interprocedural dependence analysis and parallelization. *SIGPLAN Not.*, 39(4):139–154, 2004.

[BLA07] Neil Birkbeck, Jonathan Levesque, and Jose Nelson Amaral. A dimension abstraction approach to vectorization in Matlab. In *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization*, 2007, pages 115–130. IEEE Computer Society, Washington, DC, USA.

[Cas09] Andrew Casey. Metalexer: The Lexical Specification Language. Master's thesis, Montréal, Québec, Canada, August 2009.

[CB98] Stéphane Chauveau and Franois Bodin. Menhir: An environment for high performance Matlab. In David OHallaron, editor, *Languages, Compilers, and Run-Time Systems for Scalable Computers*, volume 1511 of *Lecture Notes in Computer Science*, pages 27–40. Springer Berlin / Heidelberg, 1998. 10.1007/3-540-49530-4_3.

[CB09] Maxime Chevalier-Boisvert. McVM: An optimizing virtual machine for the MATLAB programming language. Master's thesis, Montréal, Québec, Canada, August 2009.

[CBHV10] Maxime Chevalier-Boisvert, Laurie Hendren, and Clark Verbrugge. Optimizing MATLAB through Just-In-Time Specialization. In *CC '10:*, March 2010.

[CHK93] Keith D. Cooper, Mary W. Hall, and Ken Kennedy. A methodology for procedure cloning. *Computer Languages*, 19(2):105 – 117, 1993.

[CSS99] Keith D. Cooper, Philip J. Schielke, and Devika Subramanian. Optimizing for reduced code space using genetic algorithms. In *LCTES '99: Proceedings of the ACM SIGPLAN 1999 workshop on Languages, compilers, and tools for embedded systems*, Atlanta, Georgia, United States, 1999, pages 1–9. ACM, New York, NY, USA.

[CU89] C. Chambers and D. Ungar. Customization: optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language. *SIGPLAN Not.*, 24(7):146–160, 1989.

[DB05] Hanselman D and Littlefield B. *Mastering Matlab 7*. Pearson Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2005.

[DCG95] Jeffrey Dean, Craig Chambers, and David Grove. Selective specialization for object-oriented languages. *SIGPLAN Not.*, 30(6):93–102, 1995.

[DE73] George B. Dantzig and B. Curtis Eaves. Fourier-motzkin elimination and its dual. *J. Comb. Theory, Ser. A*, 14(3):288–297, 1973.

[DRP96] Luiz De Rose and David Padua. A MATLAB to Fortran 90 translator and its effectiveness. In *ICS '96: Proceedings of the 10th international conference on Supercomputing*, Philadelphia, Pennsylvania, United States, 1996, pages 309–316. ACM, New York, NY, USA.

[DS90] Jack Dongarra and Robert Schreiber. Automatic blocking of nested loops. Technical report, Knoxville, TN, USA, 1990.

[EH07] Torbjörn Ekman and Görel Hedin. The JastAdd extensible Java compiler. *SIGPLAN Not.*, 42(10):1–18, 2007.

[FLVG95] Agustín Fernández, José M. Llaberá, and Miguel Valero-Garcia. Loop transformation using nonunimodular matrices. *IEEE Transactions on Parallel and Distributed Systems*, 6:832–840, 1995.

[FM85] J. A. B. Fortes and D. I. Moldovan. Parallelism detection and transformation techniques useful for VLSI algorithms. *Journal of Parallel and Distributed Computing*, 2(3):277 – 301, 1985.

[GDGC95] David Grove, Jeffrey Dean, Charles Garrett, and Craig Chambers. Profile-guided receiver class prediction. In *OOPSLA '95: Proceedings of the tenth annual conference on Object-oriented programming systems, languages, and applications*, Austin, Texas, United States, 1995, pages 108–123. ACM, New York, NY, USA.

[HAA+96] Mary W. Hall, Jennifer M. Anderson, Saman P. Amarasinghe, Brian R. Murphy, Shih-Wei Liao, Edouard Bugnion, and Monica S. Lam. Maximizing multiprocessor performance with the suif compiler. *Computer*, 29:84–89, 1996.

[HH97] Raymond J. Hookway and Mark A. Herdeg. Digital fx!32: combining emulation and binary translation. *Digital Tech. J.*, 9(1):3–12, 1997.

[Hof04] H.P. Hofstee. Future microprocessors and off-chip sop interconnect. *Advanced Packaging, IEEE Transactions on*, 27(2):301 – 303, may 2004.

[HU96] Urs Hölzle and David Ungar. Reconciling responsiveness with performance in pure object-oriented languages. *ACM Trans. Program. Lang. Syst.*, 18(4):355–400, 1996.

[JB07] Pramod G. Joisha and Prithviraj Banerjee. A translator system for the matlab language: Research articles. *Softw. Pract. Exper.*, 37(5):535–578, 2007.

[Kan87] Ravi Kannan. Minkowski's convex body theorem and integer programming. *Mathematics of Operations Research*, 12(3):415–440, 1987.

[KPW+07] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. Optimistic parallelism requires abstractions. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, San Diego, California, USA, 2007, pages 211–222. ACM, New York, NY, USA.

[Lam74] Leslie Lamport. The parallel execution of DO loops. *Commun. ACM*, 17(2):83–93, 1974.

[Len83] Jr. Lenstra, H. W. Integer programming with a fixed number of variables. *Mathematics of Operations Research*, 8(4):538–548, 1983.

[Li09] Jun Li. Mcfor: a MATLAB-to-Fortran 95 compiler. master's thesis. Master's thesis, Montréal, Québec, Canada, August 2009.

[LL97] Amy W. Lim and Monica S. Lam. Maximizing parallelism and minimizing synchronization with affine transforms. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, Paris, France, 1997, pages 201–214. ACM, New York, NY, USA.

[LW04] Monica S. Lam and Michael E. Wolf. A data locality optimizing algorithm. *SIGPLAN Not.*, 39(4):442–459, 2004.

[Mata] Matlab. The Language Of Technical Computing. Home page `http://www.mathworks.com/products/matlab/`.

[Matb] Matlab. The Origin of MATLAB.

[Mat02] Accelerating Matlab: The MATLAB JIT-Accelerator, 2002.

[mcl] McLab: An extensible compiler framework for Matlab.

[MHL91] Dror E. Maydan, John L. Hennessy, and Monica S. Lam. Efficient and exact data dependence analysis. *SIGPLAN Not.*, 26(6):1–14, 1991.

[Moo06] Holly Moore. *MATLAB for Engineers (ESource Series)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006.

[Par90] Review of "highly parallel computing" by g. s. almasi and a. gottlieb, benjamin-cummings publishers, redwood city, ca, 1989. *IBM Syst. J.*, 29(1):165–166, 1990. Reviewer-Lorin, Harold R.

[PEH+93] David A. Padua, Rudolf Eigenmann, Jay Hoeflinger, Paul Petersen, Peng Tu, Stephen Weatherford, and Keith Faigin. Polaris: A new-generation parallelizing compiler for MPPs. Technical report, In CSRD Rept. No. 1306. Univ. of Illinois at Urbana-Champaign, 1993.

[PH05] D.A. Patterson and J.L. Hennessy. *Computer Organization and Design: The Hardware/software Interface*. Morgan Kaufmann, 2005.

[PP94] Kleanthis Psarris and Santosh Pande. An empirical study of the I Test for exact data dependence. In *ICPP '94: Proceedings of the 1994 International Conference on Parallel Processing*, 1994, pages 92–96. IEEE Computer Society, Washington, DC, USA.

[Pug91] William Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. In *Supercomputing '91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, Albuquerque, New Mexico, United States, 1991, pages 4–13. ACM, New York, NY, USA.

[Qui84] Patrice Quinton. Automatic synthesis of systolic arrays from uniform recurrent equations. In *ISCA '84: Proceedings of the 11th annual international symposium on Computer architecture*, 1984, pages 208–214. ACM, New York, NY, USA.

[RGG+96] Luiz De Rose, Kyle Gallivan, Efstratios Gallopoulos, Bret A. Marsolf, and David A. Padua. Falcon: A MATLAB interactive restructuring compiler. In *LCPC '95: Proceedings of the 8th International Workshop on Languages and Compilers for Parallel Computing*, 1996, pages 269–288. Springer-Verlag, London, UK.

[Sar97] V. Sarkar. Automatic selection of high-order transformations in the IBM XL FORTRAN compilers. *IBM J. Res. Dev.*, 41(3):233–264, 1997.

[Sch86] Alexander Schrijver. *Theory of linear and integer programming*. John Wiley & Sons, Inc., New York, NY, USA, 1986.

[Sho81] Robert Shostak. Deciding linear inequalities by computing loop residues. *J. ACM*, 28(4):769–779, 1981.

[TAH10] Anton Dubrau Toheed Aslam, Jesse Doherty and Laurie Hendren. Aspectmatlab: An aspect-oriented scientific programming language. In *AOSD '10:Proceedings of 9th International Conference on Aspect-Oriented Software Development*, March 2010, pages 181–192.

[TWFO09] Georgios Tournavitis, Zheng Wang, Björn Franke, and Michael F.P. O'Boyle. Towards a holistic approach to auto-parallelization: integrating profile-driven parallelism detection and machine-learning based mapping. *SIGPLAN Not.*, 44(6):177–187, 2009.

[Wal88] D. R. Wallace. Dependence of multi-dimensional array references. In *ICS '88: Proceedings of the 2nd international conference on Supercomputing*, St. Malo, France, 1988, pages 418–428. ACM, New York, NY, USA.

[Wha00] John Whaley. A portable sampling-based profiler for java virtual machines. In *JAVA '00: Proceedings of the ACM 2000 conference on Java Grande*, San Francisco, California, United States, 2000, pages 78–87. ACM, New York, NY, USA.

[WL91] M. E. Wolf and M. S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Trans. Parallel Distrib. Syst.*, 2(4):452–471, 1991.

[Wol90] Micheal Wolfe. *Optimizing Supercompilers for Supercomputers*. MIT Press, Cambridge, MA, USA, 1990.

[ZCS03] Min Zhao, Bruce Childers, and Mary Lou Soffa. Predicting the impact of optimizations for embedded systems. *SIGPLAN Not.*, 38(7):1–11, 2003.

[ZWG+97] Xiaolan Zhang, Zheng Wang, Nicholas Gloy, J. Bradley Chen, and Michael D. Smith. System support for automatic profiling and optimization. *SIGOPS Oper. Syst. Rev.*, 31(5):15–26, 1997.