# McFLAT: A Profile-based Framework for MATLAB Loop Analysis and Transformations*

Amina Aslam and Laurie Hendren
amina.aslam@mail.mcgill.ca hendren@cs.mcgill.ca

School of Computer Science, McGill University, Montreal, QC, Canada

**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 paper 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.

## 1 Introduction

MATLAB is an important programming language for scientists and engineers [17]. Although the dynamic nature and lack of static type declarations makes it easy to define programs, MATLAB programs are often difficult to optimize and parallelize. The McLAB system [2] is being defined to provide an open and extensible optimizing and parallelizing compiler and virtual machine for MATLAB and extensions of MATLAB such as ASPECTMATLAB [7]. As an important part of McLAB, we are developing a framework for loop dependence tests and loop transformations, McFLAT, which is the topic of this paper.

* This work was supported, in part, by NSERC.

Due to the dynamic nature of MATLAB, there is very little static information about array dimensions and loop bounds. Furthermore, many of the scientific codes written in MATLAB can be applied to very different sized data sets. Thus, our design of McFLAT is based on a profiling phase which collects information about loop bounds over many different runs. We then have a heuristic engine which identifies important loop bound ranges and then a specializer which produces specialized code for each important range. The specializer applies loop dependence tests and loop transformations specific to the input range. Currently, for each important range, we exhaustively generate all legal specializations, but the ultimate goal is to combine this framework with a machine learning approach which will automatically generate a good specialization for the given range.

This paper describes our initial design and implementation of McFLAT and provides some exploratory experimental data obtained by using McFLAT to generate different versions of code which we execute on four different systems, Mathworks' MATLAB implementation (which includes a JIT), the GNU Octave open-source interpreter [1], our McVM interpreter and our McVM JIT [13]. Interestingly, this shows that different optimizations are beneficial for different ranges and on different MATLAB execution engines. This implies that specialization for both the range and intended execution engine is a good approach in the context of MATLAB.

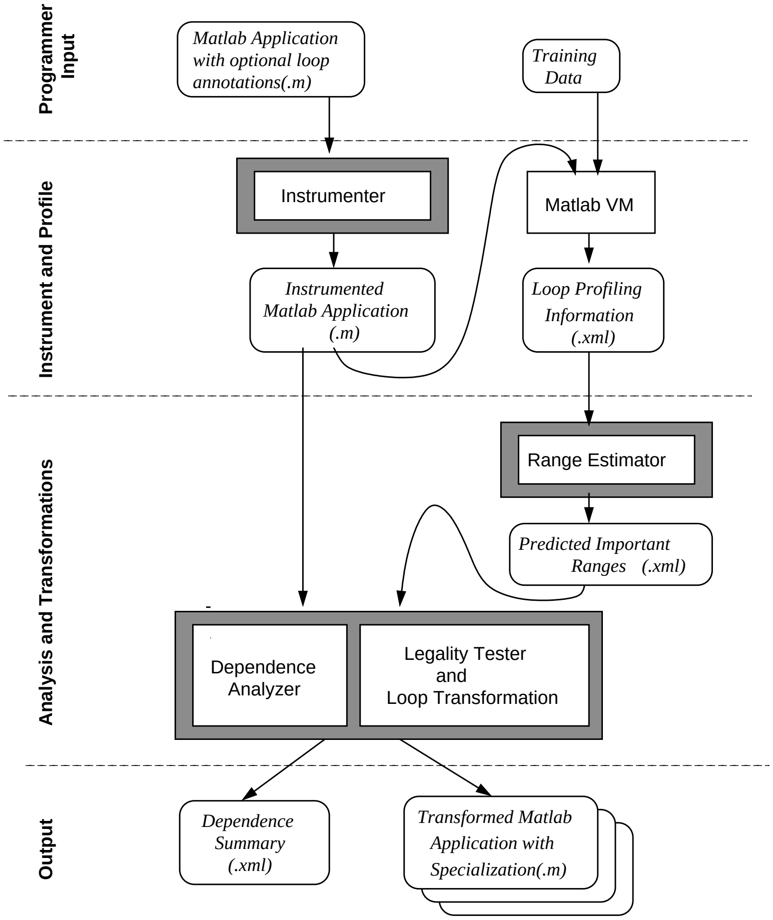
The remainder of this paper is organized as follows. In Section 2 we give a high-level view of McFLAT, and in Section 3 we provide more details of each important component. We apply our framework to a selection of benchmarks and report on the experimental results in Section 4. Finally, we discuss related work in Section 5 and conclude in Section 6.

## 2 Overview of Our Approach

The overall structure of the McFLAT framework is outlined in Figure 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 1). The output of the system is a collection of specialized programs (bottom of Figure 1), where each specialized program has a different set of transformations 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 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.



Programmer Input
Matlab Application with optional loop annotations(.m)
Training Data
Instrument and Profile
Instrumenter
Matlab VM
Instrumented Matlab Application (.m)
Loop Profiling Information (.xml)
Analysis and Transformations
Range Estimator
Predicted Important Ranges (.xml)
Dependence Analyzer
Legality Tester and Loop Transformation
Output
Dependence Summary (.xml)
Transformed Matlab Application with Specialization(.m)

**Fig. 1.** Structure of the McFLAT Framework.

## 3 Important Components of McFLAT

In this section we provide an overview of the key components of our McFLAT framework, and we briefly discuss parallel loop detection and some current limitations of the framework.

### 3.1 Instrumenter

As illustrated in the phase labeled Instrument and Profile in Figure 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 we inject instrumentation to gather, 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, and a list of variables that are written to in the loop body.

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.

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

### 3.2 Range Estimator

The *Range Estimator* is the first important component of the main part of McFLAT, the Analysis and Transformations phase in Figure 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.

**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 $\leftarrow$ lookup(LoopID, H) // get all observed values for loop with LoopID
sort(l)
importantRanges $\leftarrow$ empty
R $\leftarrow$ 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) $\geq$ threshold  then
            PredVal $\leftarrow$ maxval(sR)
            add PredVal to importantRanges
        end if
    end for
end for
return(importantRanges)
```

We determine the regions and subregions as illustrated in Figure 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 maximum observed value from the region is added to the list of important ranges.

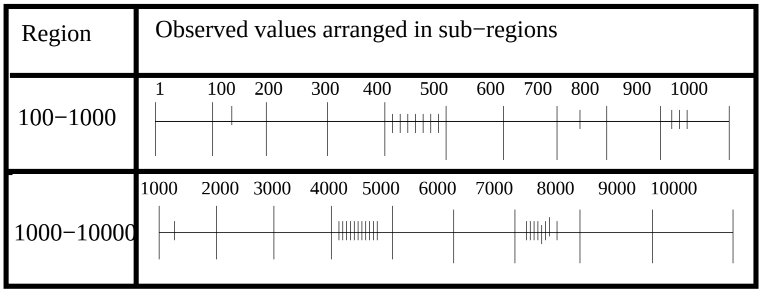


**Fig. 2.** Pictorial Example of Ranges and Subranges

### 3.3 Dependence Analysis

During this phase, McFLAT calculates dependences between all the statements in the loop body against all the predicted important ranges for that loop. It maintains various data structures supporting dependence analysis. This information is used in subsequent loop transformation phases.

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 [4, 21].

Since we have identified the upper loop bounds via our profiling, we have chosen very simple and efficient dependence testers: the *Banerjee's Extended GCD(Greatest Common Divisor) test* [8] and the *Single Variable Per Constraint Test* [4]. Currently, we have found these sufficient for our small benchmarks, but we can easily add further tests as needed.

### 3.4 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 following loop transformations that have been shown to be useful for two important goals, parallelism and efficient use of memory hierarchy [15]: *loop interchange* and *loop reversal*. For automatic detection and application of above mentioned loop transformations, we use the unimodular transformation model presented in [20]. Loop interchange and reversal are modeled as elementary matrix transformations, combinations of these transformations can simply be represented as product of elementary transformation matrices. An elementary transformation or a compound transformation is considered to be legal if the transformed distance vectors are lexicographically positive.

Apart from automatically testing the legality of loop interchange and reversal, our framework supports a larger set of transformations which can be specified by the user. This allows us to use our system as a test bed 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. 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*, *loop fusion*, *loop interchange* and *loop reversal*.

### 3.5 Parallelism Detection

Efficient parallelization of a sequential program is a challenging task. Currently our McFLAT framework automatically detects whether a **for** loop can be automatically converted to a **parfor** loop or not. The framework performs parallelization tests on the loops based on the dependence information calculated in the dependence analysis and instrumentation phase. A loop is classified as a parallel loop according to MATLAB's semantics [17], since the generated code is targeted for the MATLAB system. Thus, a loop is classified 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.
- Within the list of indices for the arrays accessed in the loop, exactly one index involves the loop index variable.
- Other variables used to index an array should remain constant over the entire execution of the loop. The loop index variable cannot be combined with itself to form an index expression.
- Loop index variables must have consecutively increasing integers.
- The value of the loop index variable should not be modified inside the loop body.

### 3.6 Current 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.

## 4 Experimental Results

In this section we demonstrate the use of McFlat through two exploratory performance studies on a set of Matlab benchmarks. Our ultimate goal is to integrate McFlat with a machine learning approach, however these example studies provide some interesting initial data. The first study examines performance and speedups 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 automatically introduce **parfor** constructs.

### 4.1 Benchmarks and Static Information

Table 1 summarizes our collection of 10 benchmarks, taken from the McLab and University of Stuttgart benchmark suites. These benchmarks have a very modest size, but yet perform interesting calculations and demonstrate some interesting behaviours. For each benchmark we give the name, description, source of the benchmark, the number of functions, number of loop nests, number of loops that can be automatically converted to parallel for loops.

| Benchmark Name | Source of Benchmark | # Lines Code | # Func. | # Loops | # Par. Loops | Benchmark Description |
|---|---|---|---|---|---|---|
| Crni | McLab Benchmarks | 65 | 2 | 4 | 1 | Finds the Crank-Nicholoson Sol. |
| Mbrt | McLab Benchmarks | 26 | 2 | 1 | 0 | Computes mandelbrot set. |
| Fiff | McLab Benchmarks | 40 | 1 | 2 | 0 | Finds the finite-difference solution to the wave equation. |
| Hnormal | McLab Benchmarks | 30 | 1 | 1 | 1 | Normalises array of homogeneous coordinates. |
| Nb1d | McLab Benchmarks | 73 | 1 | 1 | 0 | Simulates the gravitational movement of a set of objects. |
| Interpol | Uni of Stutt | 187 | 5 | 5 | 0 | Compares the stability and complexity of Lagrange interpolation. |
| Lagrcheb | Uni of Stutt | 70 | 1 | 2 | 2 | Computes Lagrangian and Chebyshev polynomial for comparison. |
| Fourier | Uni of Stutt | 81 | 3 | 3 | 2 | Compute the Fourier transform with the trapezoidal integration rule. |
| Linear | Uni of Stutt | 56 | 1 | 2 | 1 | Computes the linear iterator. |
| EigenValue | Uni of Stutt | 50 | 2 | 1 | 0 | Computes the eigenvalues of the transition matrix. |

**Table 1.** Benchmarks

### 4.2 Performance Study for Standard Loop Transformations

| Benchmark | Trans | Pred. Range 1 | | Pred. Range 2 | | Pred. Range 3 | |
|---|---|---|---|---|---|---|---|
| Name | Applied | 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% |
| | NR | **320ms** | **4.7**% | **600ms** | **6.2**% | **2.51s** | **3.4**% |
| 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 2.** Mathworks' MATLAB Execution Times and Speedups

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.

For each benchmark we ran a number of training runs through the instrumenter and profiler. For these experiments instrumented code was executed only on Mathworks' MATLAB to generate profile information. Then we used our dependence analyzer and loop transformer to generate a set of output files, one output file for each combination of possible transformations. For example, if the input file had two loops, and loop re-

| Benchmark | Trans | Pred. Range 1 | | Pred. Range 2 | | Pred. Range 3 | |
|---|---|---|---|---|---|---|---|
| Name | Applied | 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 3.** Octave Execution Times and Speedups

versal 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. For our experiments, we used a combination of both the modes that McFLAT provides for applying loop transformations i.e. *Automatic mode* and *Programmer annotated mode*.

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.

We report the results for four different MATLAB execution engines, the Mathworks' MATLAB (which contains a JIT) (Table 2), the GNU Octave interpreter (Table 3), the McVM interepreter, and the McVM JIT (McJIT) (Table 4).

|  |  | McVm(JIT) |  |  |  | McVM(Interpreter) |  |  |  |
|---|---|---|---|---|---|---|---|---|---|
| Benchmark | Trans | Pred. Range 1 |  | Pred. Range 2 |  | Pred. Range 1 |  | Pred. Range 2 |  |
| Name | Applied | Time | % Speedup | Time | % Speedup | Time | %Speedup | Time | % Speedup |
| Crni | N | **4.00s** |  | 1074s |  | 7.12s |  | 1386.2s |  |
|  | R | 4.00s | 0.0 % | **820s** | **23.6** % | **6.35s** | **10.8** % | **1341.5** | **3.2** % |
| Mbrt | N | **98.37s** |  | **675s** |  | 384s |  | 2491s |  |
|  | I | 101s | -3.3 % | 714s | -5.8% | 344s | 10.4 % | **2286s** | **8.2**% |
|  | R | 110s | -12.6 % | 781s | -15.6% | **342s** | **10.9** % | 2370s | 4.8% |
|  | (I+R) | 106s | -8.16% | 738s | -9.35% | 346s | 9.8% | 2375s | 4.6% |
| Fiff | NN | **260ms** |  | 500ms |  | 7.38s |  | 7.46s |  |
|  | RN | 260ms | -1.95% | **460ms** | **8**% | **6.95s** | **5.8**% | **7.25s** | **2.8**% |
| Hnormal | N | 5.00s |  | 8.93s |  | 7.23s |  | **11.6s** |  |
|  | R | **4.96s** | **0.8**% | **8.05s** | **10.9**% | **7.11s** | **1.6**% | 12.24s | -5.5% |
| Nb1d | N | 850ms |  | 4.10s |  | 1.41s |  | 4.24s |  |

**Table 4.** McVM Execution Times and Speedups

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.

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 lowest range and *Pred Range 3* corresponds to the highest 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 2. 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 higher data ranges.

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. Loop interchange degradation impact is less for loops that invoke a function whose value is written to an array, for example, Mbrt.

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

Now consider the execution time for Octave, given in Table 3. 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 lists the results on the subset of benchmarks currently supported. Once again loop reversal can make a significant impact on the higher ranges for the JIT, and actually also seems beneficial for the McVM(interpreter).

### 4.3 Performance study for Parallel For Loops

In Table 5 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 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.

## 5 Related Work

Of course 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 our paper.

Banerjee [9], Wolfe and Lam [20, 21] have modeled a subset of loop transformations like loop reversal, loop interchange and skewing as unimodular matrices and have devised tests to figure out the legality of these transformations. Our framework also

| Benchmark | Trans | Pred. Range 1 | | Pred. Range 2 | | Pred. Range 3 | |
|---|---|---|---|---|---|---|---|
| Name | Applied | 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 5.** Mathworks' MATLAB Execution Times and Speedups with Parallel Loops

uses 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.

Quantitative models based on memory cost analysis have been used to select optimal loop transformations [18]. Memory cost analysis chooses an optimal transformation based on the number of distinct cache lines and the number of distinct pages accessed by the iterations of a loop. Our framework 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.

A dimension abstraction approach for vectorization in MATLAB presented in [10] 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.

Another framework, presented in [22] 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 produced that represents the benefit of applying an optimization in a code context for the objective represented by the resources. McFLAT is the first step towards developing a self-learning system which would use its past experience in selecting optimal loop transformations.

### 5.1 Automatic Parallelization

Static automatic parallelism extraction have been achieved in the past [11, 16]. Unfortunately, many parallelization opportunities could still not be discovered by 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 [19]. Our approach is also based on profiling-based parallelism detection but in the context of MATLAB programming language and within the constraints of MATLAB parallel loops.

### 5.2 Adaptive Compilation

Heuristics and statistical methods have already been used in determining compiler optimization sequences. For example, Cooper et. al. [14] developed a technique using genetic algorithms to find ”good” compiler optimization sequences for code size reduction. Profile-based techniques have also been used in the past to suggest recompilation with additional optimizations. The Jalopeño JVM uses adaption system that can invoke a compiler when profiling data suggests that recompiling a method with additional optimization will be more beneficial [6]. Our work is a first step towards developing an adaptive system 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 [5], 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 MATLAB JIT-Accelerator [3], in MATLAB 6.5, that has accelerated the execution of MATLAB code. McVM [12,13] is also an effort towards JIT compilation for MATLAB, it uses function specializations based on run-time type of their arguments. The McVM(JIT) has shown performance speed-ups against MATLAB for some of our benchmarks. McFLAT, the framework presented in this paper uses profiled program features and heuristically determines loop bounds ranges to generate specialized versions of loops in the program.

## 6 Conclusions and Future Work

In this paper, 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 generated MATLAB code 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. Although the tool 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.

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 McJIT and which will decide on whether to apply a loop transformation or not depending on the benefits that the system has seen in the past. 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.

## References

1. GNU Octave. `http://www.gnu.org/software/octave/index.html`.
2. McLab: An Extensible Compiler Framework for Matlab. Home page `http://www.sable.mcgill.ca/mclab/`.
3. Accelerating Matlab, 2002. http://www.mathworks.com/company/newsletters/digest/sept02/accel_-matlab.pdf.
4. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison Wesley, 1985.
5. G. Almasi and D. A. Padua. MaJIC: A MATLAB Just-In-Time Compiler. In *Languages and Compilers for Parallel Computing*. Springer Berlin / Heidelberg, 2001.
6. M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive Optimization in the Jalapeño JVM. In *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 47–65, New York, NY, USA, 2000. ACM.
7. T. Aslam, J. Doherty, A. Dubrau, and L. Hendren. AspectMatlab: An Aspect-Oriented Scientific Programming Language. In *Proceedings of 9th International Conference on Aspect-Oriented Software Development*, pages 181–192, March 2010.

8. U. K. Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, Norwell, MA, USA, 1988.
9. U. K. Banerjee. *Loop Transformations for Restructuring Compilers: The Foundations*. Kluwer Academic Publishers, Norwell, MA, USA, 1993.
10. N. Birkbeck, J. Levesque, and J. N. Amaral. A Dimension Abstraction Approach to Vectorization in Matlab. In *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization*, pages 115–130, Washington, DC, USA, 2007. IEEE Computer Society.
11. M. G. Burke and R. K. Cytron. Interprocedural Dependence Analysis and Parallelization. *SIGPLAN Not.*, 39(4):139–154, 2004.
12. M. Chevalier-Boisvert. McVM: An Optimizing Virtual Machine for the MATLAB Programming Language. Master's thesis, McGill University, August 2009.
13. M. Chevalier-Boisvert, L. Hendren, and C. Verbrugge. Optimizing MATLAB through Just-In-Time Specialization. In *International Conference on Compiler Construction*, pages 46–65, March 2010.
14. K. D. Cooper, P. J. Schielke, and D. 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*, pages 1–9, New York, NY, USA, 1999. ACM.
15. M. S. Lam and M. E. Wolf. A Data Locality Optimizing Algorithm. In *PLDI'91: Programming Language Design and Implementation*, volume 39, pages 442–459, New York, NY, USA, 2004. ACM.
16. A. W. Lim and M. 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*, pages 201–214, New York, NY, USA, 1997. ACM.
17. Matlab. The Language Of Technical Computing. Home page `http://www.mathworks.com/products/matlab/`.
18. V. Sarkar. Automatic Selection of High-Order Transformations in the IBM XL FORTRAN compilers. *IBM J. Res. Dev.*, 41(3):233–264, 1997.
19. G. Tournavitis, Z. Wang, B. Franke, and M. F. O'Boyle. Towards a Holistic Approach to Auto-Parallelization: Integrating Profile-Priven Parallelism Detection and Machine-Learning based Mapping. In *PLDI'09: Programming Languages Design and Implementation*, volume 44, pages 177–187, New York, NY, USA, 2009. ACM.
20. 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.
21. M. J. Wolfe. *Optimizing Supercompilers for Supercomputers*. MIT Press, Cambridge, MA, USA, 1990.
22. M. Zhao, B. Childers, and M. L. Soffa. Predicting the Impact of Optimizations for Embedded Systems. In *Proceedings of the 2003 ACM SIGPLAN Conference on Language, Compiler, and Tool Support for Embedded Systems*, volume 38, pages 1–11, San Diego, CA, USA, 2003. ACM.