



Performance analyses and code transformations for MATLAB applications

Patryk Kiepas

► To cite this version:

Patryk Kiepas. Performance analyses and code transformations for MATLAB applications. Computation and Language [cs.CL]. Université Paris sciences et lettres, 2019. English. NNT : 2019PSLEM063 . tel-02516727

HAL Id: tel-02516727

<https://pastel.hal.science/tel-02516727>

Submitted on 24 Mar 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



THÈSE DE DOCTORAT
DE L'UNIVERSITÉ PSL

Préparée à MINES ParisTech

Analyses de performances et transformations de code pour les applications MATLAB

Performance analyses and code transformations for MATLAB applications

Soutenue par

Patryk KIEPAS

Le 19 decembre 2019

Ecole doctorale n° 621

**Ingénierie des Systèmes,
Matériaux, Mécanique,
Énergétique**

Spécialité

**Informatique temps-réel,
robotique et automatique**

Composition du jury :

Christine EISENBEIS Directrice de recherche, Inria / Paris 11	<i>Présidente du jury</i>
João Manuel Paiva CARDOSO Professeur, University of Porto	<i>Rapporteur</i>
Erven ROHOU Directeur de recherche, Inria Rennes	<i>Rapporteur</i>
Michel BARRETEAU Ingénieur de recherche, THALES	<i>Examineur</i>
Francois GIERSCHE Ingénieur de recherche, THALES	<i>Invité</i>
Claude TADONKI Chargé de recherche, MINES ParisTech	<i>Directeur de thèse</i>
Corinne ANCOURT Maître de recherche, MINES ParisTech	<i>Co-directrice de thèse</i>
Jarosław KOŹLAK Professeur, AGH UST	<i>Co-directeur de thèse</i>

Abstract

MATLAB is an interactive computing environment with an easy programming language and a vast library of built-in functions. Therefore, researchers in Computer Science and Engineering (CSE) often use it as a prototyping tool. However, some features of this environment, such as its dynamic language or interactive style of programming, affects how fast programs can execute. The deficiency of performance is especially visible in compute-intensive applications, such as image processing or machine learning. These applications perform computations on a massive amount of data, ideally, as fast as possible.

The goal of this thesis is to develop techniques for the analysis and optimisation of general MATLAB programs. Current methods for increasing performance of the programs include two approaches: (1) systematic code transformations applied without any consideration of their impact on the program execution, and (2) translation of MATLAB codes to static languages to benefit from years of research into optimising compilers for C and Fortran languages. While the translation of MATLAB programs (2) skips the MATLAB environment entirely, the systematic code transformation (1) does not consider the exact inner workings of this environment at all. In this thesis, we aim to fill this gap by focusing on research questions about how to analyse the black-box MATLAB environment, and what new code transformation could optimise the performance of programs without incurring additional development cost on programmers.

MATLAB environment consists of an interpreter, garbage collector, and Just-In-Time (JIT) compiler among others. However, these components are neither open-source nor documented. Therefore, the environment is a black-box which requires an entirely new approach for its analysis, as current performance modelling techniques aim mainly at open-source solutions. To address this challenge, we focus on the execution of MATLAB programs directly on the CPU. For this task, we use a well-known performance event profiles which measure how particular events on the CPU change over time, e.g. the number of cache misses or floating-point operations. Furthermore, we introduce a notion of execution regions which divides performance profiles into segments with particular properties, e.g. a data copy or a floating-point computation.

Using performance event profiles and execution regions, we have analysed how particular expressions are scheduled for the execution by the MATLAB JIT compiler. The compiler generates machine code for a batch of functions at a time, called an instruction block. Therefore, by observing the activity of an instruction cache on the CPU, we can track when each block starts and ends. An instruction block is either a set of combinable functions which coexist together or a single function requiring the whole instruction block for itself.

To predict the type and the order of instruction blocks, we have proposed a static tree-based model called the instruction tree. The model predicts, directly from the MATLAB code, what execution regions are generated while running the program. The instruction tree gives an insight into the execution of MATLAB programs, and it can be a basis for future performance models. Furthermore, with the knowledge gained from the analysis of the MATLAB environment, we have proposed several code transformations which create new optimisation opportunities. Repacking of array slices can increase the amount of JIT-compiled code in programs by performing array slicing before the actual computation, and range simplification, which reduces the number of redundant computation and indexing phases. Also, the proposed model indicates when the reordering, fusing and splitting of (sub-)expressions could increase the program performance as well.

The main contribution of this thesis is the methodology to analyse and discover how a black-box environment, such as MATLAB, executes programs. Furthermore, with the increased knowledge about the execution of programs, we have proposed several code transformations which can improve the performance of applications. During the work on the thesis, we have developed HU!M, a source-to-source compiler for MATLAB programs which performs automatic analyses and code transformations, and an interface mPAPI for accessing hardware performance counters directly from the MATLAB environment.

Acknowledgements

The creation of this thesis was an effect of superposition of various events and people met during my life. I am not exactly sure to whom I can attribute my appreciation and fascination of science, but whoever you are: well done!

Firstly, I am grateful to my supervisors Corinne Ancourt, Claude Tadonki, and Jarosław Koźlak for their helpful, supportive, and encouraging guidance. This thesis is the result of our meetings and all tough questions asked during them. I have learnt a lot from you. Thank you.

CRI laboratory is a unique mix of science, friendship, and work, where time does fly a bit differently. I express my gratitude to past and present CRI members for creating this warm atmosphere and including me in the group: Maksim Berezov, Catherine Le Caër, Fabien Coelho, Laurent Daverio, Emilio Gallego, Florian Gouin, Pierre Guillou, Olfa Haggui, Olivier Hermant, François Irigoien, Pierre Jouvelot, Claire Medrala, Benoît Pin, Bruno Sguerra, Lucas Sguerra, Pierre Wargnier, and Katarzyna Węgrzyn-Wolska. I am indebted to François and Fabien for making my last year of studies possible.

I thank Michel Barreteau, François Giersh and Frédéric Barbaresco from THALES for their cooperation and an opportunity to prepare this thesis in the first place. Our meetings, although sparse, were always entertaining.

I say thanks to my dense friends: Jakub and Ziejka for always being there on-line; Panek for coming to us off-line; Adilla for all the voices; Gabriella for many topics to think about; Nathan for speed hiking; Laila for always having nuts; Tuanir for remembering the sound of bongos; Corinne for making Monika and me the happiest dog sitters of Orson on this side of the Seine; and the Chileans: Rocío, Estaban, and Hector for crossing our timelines.

Finally, I send love to my whole family for years of cheering, especially to my parents, Ewa and Arkadiusz, for all the support. I thank my sister Kinia and Leszek, for the best B&B in Warsaw. I am grateful to Monika's parents, Małgorzata and Witold, for the help and time spent. At last, I send my love to Monika for not falling asleep while listening about MATLAB. You are, indeed, optimal.

Contents

1	Introduction	15
1.1	Motivation	16
1.2	Research challenges	18
1.3	Thesis contributions	19
1.4	Thesis structure	20
2	Related work	23
2.1	Acceleration of MATLAB programs	24
2.1.1	Compilation of MATLAB programs	24
2.1.2	Transformation of MATLAB code	27
2.1.3	Alternative execution environments	28
2.1.4	Analysis of MATLAB programs	28
2.2	Performance analysis	31
2.2.1	Metrics and models	31
2.2.2	Hardware performance counters	32
2.2.3	Profiling	34
2.2.4	Performance of execution environments	35
2.3	Conclusion	35
3	Performance event profiles	37
3.1	Overview	38
3.2	Motivation	39
3.3	Building performance profiles	41
3.3.1	Selecting the sampling event	43
3.3.2	Selecting the sampling threshold	45
3.3.3	Performance profiles with mPAPI	48
3.4	Finding execution regions	50
3.5	Case study: cost of array slicing	52
3.6	Conclusion	55
4	Execution model for MATLAB	57
4.1	Scope of the execution model	58
4.2	Instruction blocks in JIT compilation	59

4.3	Detecting instruction blocks	61
4.4	JIT compilation of functions	64
4.4.1	Built-in functions	64
4.4.2	User-defined function	66
4.5	Instruction tree	68
4.5.1	Building minimal instruction tree	69
4.5.2	Predicting execution from minimal instruction tree . .	72
4.6	Conclusion	74
5	Code transformations for array operations	77
5.1	Redesigning array slicing	78
5.1.1	Dynamic array slicing	78
5.1.2	Eliminating redundant 0-initialisation	79
5.2	Repacking of array slices	82
5.3	Range simplification	83
5.4	Profile-guided loop vectorisation	86
5.5	Conclusion	91
6	HU!M compiler	93
6.1	Overview	94
6.2	Influences	95
6.3	Code analysis	97
6.4	Code transformation	100
6.4.1	Loop vectorisation	100
6.4.2	Fast array slicing substitution	102
6.4.3	Repacking of array slices	103
6.5	Conclusion	103
7	Evaluation of the execution model and code transformations	105
7.1	Evaluation of the execution model	106
7.1.1	Model precision	106
7.1.2	Splitting expressions	106
7.1.3	Reordering operations	108
7.1.4	Information limit of performance profiles	110
7.2	Evaluation of the range simplification	111
7.3	Evaluation of the repacking of arrays	114
7.4	Conclusion	116
8	Conclusion	119
8.1	Summary	120
8.2	Future work	121

A	Experiment methodology	125
A.1	Preparation of the environment.	125
A.2	Collecting measurements	126
A.3	Machine specification	126
B	mPAPI	129
B.1	mPAPI interface.	130
B.1.1	Enumerating available performance events	130
B.1.2	Measuring performance events in counting mode . . .	130
B.1.3	Measuring performance events in sampling mode . . .	131
C	Menchi	133
C.1	Benchmark preparation	134
C.2	Experiment specification	135
C.3	Experiment modes	136
D	Accompanying materials	141

List of Figures

1.1	Multiplication of random square matrices in C, MATLAB, and Python	16
1.2	Improving performance of MATLAB	17
3.1	Components of program execution in MATLAB	38
3.2	Performance event profiles for the <i>striad</i> kernel	40
3.3	Performance profiles built using various sampling events . . .	44
3.4	Performance profiles built using various sampling thresholds .	46
3.5	Impact of the sampling threshold on the duration of performance profiles	47
3.6	Workflow of creating performance traces with mPAPI	49
3.7	Cost of performing data copy during array slicing	54
4.1	Components of MATLAB expressions with array operations .	58
4.2	Detection of instruction blocks with code examples	63
4.3	Code examples for testing dynamic compilation of user-defined functions	67
4.4	Step 1: Conversion of an expression to AST	69
4.5	Step 2: Conversion from an AST to the instruction tree . . .	70
4.6	Step 3: Removal of array reference leaves	70
4.7	Step 4: Merging of instruction blocks to form the minimal instruction tree	72
4.8	Instruction chain obtained from the flattened instruction tree	74
4.9	Prediction of the execution regions and their order	75
5.1	Zero initialisation in array slicing	80
5.2	Results of applying repacking of array slices	84
5.3	Performance profiles of array slicing with ranges	85
5.4	Profiling of loop vectorisation	88
5.5	Results of profile-guided vectorisation	90
6.1	Overview of analyses in HU!M	97
6.2	Hierarchy of classes representing the instruction tree	97
6.3	Overview of transformations in HU!M	100

6.4	Fast array slicing substitution in HU!M	102
6.5	Repacking of array slices in HU!M	103
7.1	Execution regions of MIT-2 example	107
7.2	Example of splitting an expression into sub-expressions	108
7.3	Splitting expression to perform vector operations	109
7.4	Minimal instruction tree after reordering of instructions . . .	109
7.5	Reordering expression to reduce instruction blocks	110
7.6	Execution regions of MIT-1 example	111
7.8	Performance profiles of array slicing with ranges	113
7.9	Execution time of range simplification	114
7.10	Repacking of array slices on Livermore kernels	115
7.11	Repacking of array slices on LCPC16 kernels	117
A.1	Hierarchical topology of test machines	127
C.1	Workflow of code benchmarking with Menchi	133
D.1	Optimised loop crni2 from Chen et al. [1]	141
D.1	Optimised loops nw2 and nw3 from Chen et al. [1]	142
D.1	Optimised loop fft1 from Chen et al. [1]	143
D.2	Extended version: cost of performing data copy during array slicing	144
D.3	Execution regions of MIT-3 example	145
D.4	Execution regions of MIT-4 example	145
D.5	Execution regions of MIT-5 example	146
D.6	Execution regions of MIT-6 example	146
D.7	Execution regions of MIT-7 example	147
D.8	Execution regions of MIT-8 example	147

List of Tables

2.1	List of research and industrial compilers for MATLAB.	25
3.1	Size of profile files with changing sampling threshold	48
3.2	Selected programs for the analysis of data copy cost	53
4.1	Performance event candidates for the detection of instruction blocks	62
4.2	Test codes used for the detection of dynamic compilation . . .	65
4.3	List of <i>single</i> and <i>combinable</i> built-in functions	66
4.4	Examples of expressions with their minimal instruction trees .	73
5.1	Experiment reproduction of vectorisation by Chen et al. [1] .	87
5.2	Results of loop profiling for profitable loop vectorisation . . .	89
7.1	Machine instructions counted by the <code>FP_ARITH_INST_RETIRED:*</code> family of performance events on Skylake and Coffee Lake microarchitectures.	112
A.1	Specification of test machines	126
C.1	Benchmark specification generated by <code>CodeExtractor</code>	135
C.2	Experiment modes in <code>Menchi</code> tool	137

Chapter 1

Introduction

Résumé

Pour de nombreux chercheurs et programmeurs, MATLAB est un outil pratique pour le prototypage rapide de solutions à des problèmes de calculs complexes, car le langage possède une syntaxe claire et une vaste bibliothèque de fonctions et d'algorithmes intégrés. Bien que les versions récentes de MATLAB soient compilées en temps réel (JIT), ses performances sont souvent inférieures à celles d'autres langages tels que le C, Fortran et même Python. Auparavant, les chercheurs ont adopté deux approches pour améliorer les performances des programmes MATLAB : 1) la compilation du code MATLAB comme celle des langages C et Fortran, et 2) la transformation du code des programmes MATLAB pour utiliser des constructions de langage plus rapides. Alors que 1) la compilation ignore entièrement l'environnement d'exécution de MATLAB, 2) la transformation du code ne fonctionne bien que lorsque nous savons précisément comment MATLAB exécute les programmes. Dans cette thèse, nous présentons une analyse de performance permettant de découvrir comment les applications s'exécutent dans l'environnement MATLAB. De plus, nous formalisons l'exécution des programmes avec un modèle d'exécution basé uniquement sur les arbres permettant d'effectuer de nouvelles transformations de code qui augmentent les performances des programmes MATLAB compilés en temps réel (JIT). Nous avons également développé plusieurs outils et un compilateur qui facilite l'utilisation de l'analyse des performances et l'application des transformations de code.

Introduction

For many programmers and researchers, MATLAB is an everyday tool, capable of expressing even the most complex computational problems in an easy, concise, and interactive way using its dynamic execution environment. MATLAB achieves this by mastering the idea of programming language as mathematical notation, first proposed by Iverson [2] in his APL language. Consequently, MATLAB is a domain-specific language dedicated to linear algebra and array-based computations. Moreover, in the language, vectors,

<pre> 1 // C language 2 double *randmatmul(int n) { 3 double *A = myrand(n*n); 4 double *B = myrand(n*n); 5 double *C = (double*) 6 malloc(n*n*sizeof(double)); 7 cblas_dgemm(CblasColMajor, 8 CblasNoTrans, CblasNoTrans, n, n, 9 n, 1.0, A, n, B, n, 0.0, C, n); 10 free(A); 11 free(B); 12 return C; 13 }</pre>	<pre> 1 # Python + NumPy 2 import numpy as np; 3 def randmatmul(n): 4 A = np.random.rand(n,n) 5 B = np.random.rand(n,n) 6 return np.matmul(A,B) 1 % MATLAB 2 function C = randmatmul(n) 3 A = rand(n,n); 4 B = rand(n,n); 5 C = A * B; 6 end</pre>
---	---

Figure 1.1: Code examples for the multiplication of random square matrices written in three different languages: C, MATLAB, and Python (from Julia Microbenchmarks [3]). C code requires manual memory management, an explicit call to the external BLAS routine, and type declarations.

matrices and multi-dimensional arrays are first-class citizens with dedicated operations. Thus, expressing computations using these multi-element structures is as easy as working with scalar variables in any other language.

MATLAB is not only a programming language, it is a problem-solving environment [4], with more than 2900 built-in functions, 53 specialised tool-boxes, 8 compilers, and 1 code profiler (as of version R2019b). Figure 1.1 compares multiplication of random square matrices written in C, Python, and MATLAB. While the code of Python with NumPy library is fairly similar to the MATLAB code, the C code requires an implementation of function `myrand` for random generation of arrays (MATLAB has `rand`); a call to the external `dgemm` BLAS routine for matrix multiplication (MATLAB uses Intel MKL [5] for `*` operator seamlessly); a manual memory management using `malloc` and `free` (MATLAB performs automatic memory [de]allocation); and explicit declarations of type `double` (MATLAB works on `double` type by default, also it has dynamic and weak type system). The inherent easiness of writing code together with the numerical orientation make MATLAB especially suitable for fast prototyping of applications in computational science and engineering (CSE), signal processing, control systems, image processing, machine learning, and many more disciplines.

1.1 Motivation

The interactive and dynamic natures of MATLAB, so desirable for fast prototyping, required MATLAB programs to be interpreted at first in 1984 [6], and then gradually moved to more powerful techniques such as program

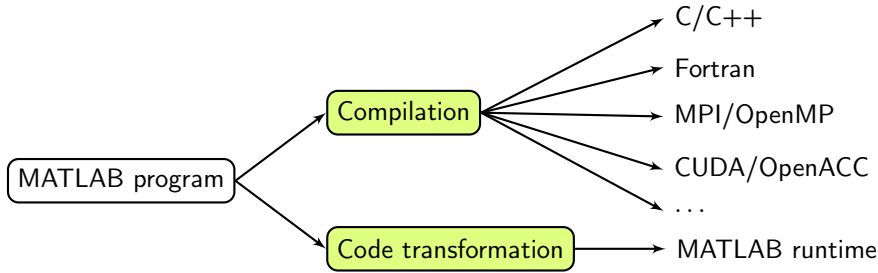


Figure 1.2: Two possible approaches to improving performance of MATLAB programs.

execution using bytecode-based virtual machine (VM) [7] and Just-In-Time (JIT) compilation [8]. In order to stay relevant and increase its performance, over the years, MATLAB received support for vector instructions around year 2000 [9], included Just-In-Time (JIT) compiler in 2002 with MATLAB 6.5, introduced explicit parallelism with Parallel Computing Toolbox in 2004, executes built-in functions on many-threads since R2007b, included computation on Graphics Processing Unit (GPU) in R2010b, and finally, obtained new execution engine (LXE) in R2015b [10] which combines the interpreter and the JIT compiler under a single monolithic architecture.

Unfortunately, the JIT compilation induces additional costs related to the runtime preparation of the code just before the execution. In order to compete with statically compiled languages like C and Fortran, the JIT compiler must be as good, or even better, as compilers for those languages. Therefore, since the beginning of MATLAB, researchers and companies have worked on methods and tools for optimising MATLAB programs [1, 11–21].

Figure 1.2 presents two common approaches to improving MATLAB performance: (1) compilation and (2) code transformation. Other techniques include, e.g. developing new virtual (VM) machines [22], JIT compilers [23], and parallel extensions [24, 25]. From the previous works, the MATLAB translation to C and Fortran are, by far, the most popular approaches, probably because they bypass performance limitations of the MATLAB runtime whatsoever. Instead, the compilation can benefit from years of research on optimising compilers [26–28]. However, compilation of MATLAB programs breaks the fast prototyping development cycle. Imagine our problem is compute-intensive and it requires compilation before each run. In such case, we need to leave the MATLAB environment and switch to other compilation toolchain. Therefore, in order to facilitate program development, we have focused on source-to-source transformations of MATLAB programs. Moreover, code transformations for MATLAB were much less investigated by researchers than MATLAB compilation, and as such, we believe there is still room for improvement.

1.2 Research challenges

Researchers have applied code transformations to MATLAB since the 1990s starting with the FALCON project [29, 30]. Around year 2000, the huge benefit of loop vectorisation made it a primary code optimisation technique for MATLAB programmers. Since then, researchers applied vectorisation automatically to MATLAB programs [9, 21]. However, because of the improvements to JIT compilation in MATLAB, the systematic vectorisation of every loop is no longer a viable approach [1, 31]. Therefore, the successful application of loop vectorisation and any other code transformation requires the understanding of how exactly MATLAB executes programs and a careful consideration of when and what transformation to apply.

MATLAB is a closed source, proprietary environment. Thus, programmers and researchers have almost no details from the vendor about how MATLAB really works. Moreover, the MATLAB language is dynamic with a weak type system which is not the best input for static code analysis, because of the lack of many information about the source code. This situation leads to the need of developing new approaches of characterising program execution and MATLAB itself. For this task, we have previously worked on several research directions, unfortunately with no positive results apart from gaining new knowledge.

Static heuristics for loop vectorisation. Our first approach to improving performance of MATLAB programs was to create heuristics for checking when to apply loop vectorisation. In order to find key elements of programs contributing the most to the performance of loop vectorisation, we have used several performance models, analyses and metrics such as the Roofline Model [32], Top-Down Microarchitectural Analysis [33], cache and data reuse models, monitoring of hardware performance counters in the counting mode. However, every tested approach gave us only a global view on the program execution without any consideration for time-varying behaviour, which turned out to be a key factor. The lack of results from this approach lead us to a direction of profile-guided vectorisation using a simple profiling scheme described in Section 5.4 and [31]. Although, very useful and powerful, profile-guided vectorisation is not a solution to the initial goal of having static heuristics.

Machine Learning model for selecting loop vectorisation. Our next approach to the previous problem was based on Machine Learning for building a decision model for loop vectorisation. For this task, we have examined two approaches with vastly different representations of MATLAB programs. The work of Cavazos et al. [34] uses a vector of values from hardware performance counters to encode a single code example. Later, in order to predict if a loop is worth vectorising, we just run the loop and collect performance counters

which were then feed to the model. In contrast, the work of Cummins et al. [35] uses fully static representation, the string of the code example encoded using techniques used also in Natural Language Processing (NLP). Although, using the approach by Cavazos et al. [34] we were able to create promising models with high precision, we failed too find in them any important insights.

Warm-up state of JIT compiler in MATLAB. In this analysis, we have focused on the JIT compiler entirely and followed a study of Barrett et al. [36] about characterising warm-up stages of popular virtual machines (VM), e.g. HHVM [37], Graal [38], HotSpot JVM [39], and others. In the study, various benchmarks were repeatedly executed on each virtual machine to see when the JIT compilation happens and how it affects the performance. In MATLAB, we have observed that JIT compilation usually increases the program performance, however, in very rare cases it might even lead to performance degradation. Moreover, we have observe the JIT compilation happens during the first run of a program, which indicates that the JIT compiler in MATLAB has a single-tier policy [40].

All abovementioned challenges have only emphasized the need for a different approach to analyse, model, and understand how MATLAB executes programs, before working on code transformations for them.

1.3 Thesis contributions

In the thesis, we have focused on the development of new analyses and code transformations for MATLAB programs. The main contributions are built around three research questions:

- **RQ1:** Can we analyse and model how MATLAB works seen as a black-box?
- **RQ2:** Are we able to propose new code transformations for MATLAB?
- **RQ3:** Is it possible to improve MATLAB performance without breaking much the fast prototyping cycle?

Analysis and modelling. The answer to question **RQ1** is divided into two Chapters 3 and 4. In Chapter 3, we introduce *performance event profiles* (PEP) which are an inventive use of hardware performance counters to analyse time-varying aspects of the program execution. Later, in Chapter 4, we use the profiles to discover and build an execution model for MATLAB expressions. The model is encoded as *instruction trees* where each node represents a block of instructions executed together by the JIT compiler. Moreover, connections between these nodes indicate their execution order. The result of the model is a prediction of how MATLAB executes a given expression.

New code transformations. To answer the next question, **RQ2**, we use the combined results of Chapters 3 and 4 to propose several specialised code transformations for MATLAB programs, e.g. *repacking of array slices* and *range simplification* in Chapter 5. For example, the repacking leverages the fact that JIT compiler in MATLAB prefers expressions without indexed variables, because then the expressions can be compiled as one block of instructions. Therefore, the repacking extracts and substitutes indexed variables with references to temporary arrays. Furthermore, in Section 5.4, we take a fresh look on loop vectorisation applied to MATLAB programs in the profile-guided context.

Fast prototyping cycle. The last question **RQ3** is answered indirectly with the knowledge from previous answers and the content of Chapter 6. The chapter presents our **HU!M** compiler which implements presented execution model and code transformations. These transformations not only keep the similar level of code readability, but also once applied, they persist in the code without future need to reapply them again, as opposed to MATLAB compilation which works on the whole compilation unit (usually a function) and requires recompilation even after small change in the source code. Moreover, the majority of presented code transformations (except for loop vectorisation) requires little to none of code analysis, while other, e.g. *dynamic array slicing*, postpone the analysis until the runtime.

1.4 Thesis structure

In Chapter 2, we outline collection of works related to three topics: (1) acceleration of MATLAB programs; (2) performance analysis of computer programs; and (3) analysis of runtime environments. The topic (1) shows the scale of the research that went into the optimisation of MATLAB programs. Moreover, topics (2) and (3) show other attempts to understand the behaviour of program execution and runtime environments, e.g. Python and JVM.

The next three Chapters 3 to 5 describe our contributions:

- *Performance event profiles* (PEP) are an inventive way to use performance profiles built using hardware performance counters to discover and understand the program execution (Chapter 3). Every performance profile consists of several *execution regions*, each depicting different kind of a computation. Furthermore, we present how to create performance profiles using our open source tool **mPAPI**.
- Execution model of MATLAB expressions is encoded as an *instruction tree* which groups instructions into blocks (Chapter 4). Each *instruction block* is separately compiled for the execution by the JIT compiler. Using our model, we are able to predict the order of instruction blocks and

their content, by tracking which instructions merge together in the process of obtaining the *minimal instruction tree*.

- *Dynamic array slicing, repacking of array slices, range simplifications* are code transformations for MATLAB programs introduced in Chapter 5. They improve performance of MATLAB expressions without impacting drastically the code readability. Moreover, in Chapter 5, we have analysed loop vectorisation applied with profiling information.

In Chapter 6, we present our source-to-source compiler **HU!M** capable of applying transformations from Chapter 5 in an automatic manner. Moreover, the compiler implements our execution model from Chapter 4, and it is capable of giving prediction about the program execution directly from an input source code. Finally, Chapter 8 summarises the research tasks performed during the work on the thesis, obtained results, and the future work.

The thesis consists of 4 appendix with descriptions of experiment methodology, our two tools **mPAPI** and **Menchi**, and a collection of additional results. Appendix A describes the experiment methodology used for every results (plots included) in the thesis, except for the experiments in Section 5.4. The goal of the methodology is to limit measurement errors and non-deterministic events as much as possible, and to correctly summarise the obtained measurements, e.g. using confidence intervals [36, 41, 42]. Moreover, Appendix A contains a list of two machines used in our experiments.

Appendix B describes **mPAPI** which is a MATLAB interface for accessing hardware performance counters built on top of the PAPI library [43]. The interface extends the capabilities of the official PAPI interface for MATLAB with collecting performance counters in the sampling and the multiplexed modes, and allowing to measure counters for separate threads.

In Appendix C, we describe **Menchi**, an experiment generator tool which was used to prepare each test in the thesis in a single and consistent manner.

Finally, Appendix D contains additional codes and plots which could not fit into the main part of the manuscript.

Chapter 2

Related work

Résumé

Dans ce chapitre, nous nous sommes concentrés sur les travaux de recherche liés à l'accélération des programmes MATLAB et à l'analyse des performances des applications. Bien qu'il existe plusieurs autres domaines de recherche proches des travaux présentés dans cette thèse, ces deux sujets sont les plus importants.

Au fil des ans, les chercheurs ont créé plusieurs compilateurs de MATLAB vers d'autres langages : FALCON, Otter, Match, MATISSE, ou le codeur officiel de MATLAB, pour n'en citer que quelques-uns. Aujourd'hui, seul MATLAB Coder est prêt à être utilisé par les programmeurs et les scientifiques, mais le processus d'utilisation n'est pas entièrement automatisé. D'autre part, la recherche sur les transformations de code des programmes MATLAB est très rare. Pendant de nombreuses années, le savoir collectif pour augmenter les performances a inclus la vectorisation des boucles dans les codes MATLAB. Cependant, les compilateurs récents de MATLAB peuvent produire un code plus rapide sans vectorisation. La simple décision de vectoriser une boucle ne peut être prise qu'avec l'aide de modèles d'exécution qui ont été étudiés auparavant dans le contexte de MATLAB.

Pour créer un modèle d'exécution et un modèle de performance, nous devons utiliser des techniques d'analyse de performance. Les mesures et modèles de performance classiques, tels que le nombre d'instructions par cycle (IPC) ou le modèle de cache d'exécution (ECM), soit ne prennent pas en compte le comportement variable dans le temps des programmes analysés, soit ne décrivent qu'une perspective abstraite de l'exécution du programme. Les compteurs de performance matérielle donnent une vue détaillée de l'exécution du programme car ils représentent la façon dont une unité centrale exécute le programme. De plus, les valeurs des compteurs de performance peuvent être échantillonnées dans le temps, donnant ainsi une description de l'exécution du programme qui varie dans le temps. La majorité des analyses de performance ont été appliquées à des langages et des environnements d'exécution de logiciels libres, par opposition aux boîtes noires de logiciels propriétaires comme MATLAB.

Introduction

The work presented in the thesis, spans across several important topics: programming languages, compiler construction, performance analysis, design of processors, code transformations, compilation, to name a few. For this chapter, we have selected two most important topics connected to our work: (1) acceleration of MATLAB programs; (2) performance analysis of computer programs and runtime environments.

2.1 Acceleration of MATLAB programs

Acceleration of MATLAB programs has a long history full of many research and industrial projects. In this section, we describe four major components of improving MATLAB programs: (1) compilation; (2) transformation; (3) development of new virtual machines and parallel extensions; and (4) analysis.

2.1.1 Compilation of MATLAB programs

Compilation of MATLAB programs, especially to static languages like C or Fortran, brings the results of many research works devoted to loop transformations and optimising compilers for these languages. Moreover, because the input language is MATLAB, programmers still benefit from MATLAB language clarity and fast prototyping properties. Furthermore, compilation of MATLAB is a solution for porting programs to parallel and heterogeneous computing platforms.

Unfortunately, not all MATLAB built-in functions are supported by MATLAB compilers. Even today, the official MATLAB to C compiler, **MATLAB Coder**, supports the majority but not all built-in functions [44]. Moreover, other MATLAB compilers support only a handful of built-ins. Although, the compilation of MATLAB programs is common and beneficial, it is not always applicable. However, some compilers port a MATLAB program onto a new platform which does not run MATLAB environment. A good example is the MATISSE compiler which targets embedded systems [45] and heterogeneous computing platforms with OpenCL [46], otherwise not available for MATLAB programs.

In this section, we present various compilers for MATLAB language depicted in Table 2.1. Moreover, we briefly describe some of them in the subsequent paragraphs to illustrate the variety of targeted languages and architectures. However, we do not focus on their code analysis capabilities, leaving this part for the Section 2.1.4.

FALCON. FALCON is a programming infrastructure for creating numeric applications using MATLAB language [48]. The tool is able to transform MATLAB code, as well as compiled it to Fortran 90. FALCON infers

Table 2.1: List of research and industrial compilers for MATLAB.

Compiler	Active years	Target language	Platform	Ref
FALCON	1995-2001	Fortran/MATLAB	CPU	[11, 47, 48]
Otter	1998-1999	C/MPI	CPU	[12, 49]
Menhir	1998-1999	C/Fortran	CPU	[19, 50]
MATCH	1999-2003	VHDL	FPGA/DSP chips	[51-53]
Mat2C	2006-2007	C	CPU	[54]
OMPC	2009-2010	Python	CPU	[55]
MEGHA	2011-2012	C++/CUDA	CPU/GPU	[56]
MATISSE	2013-2017	C/OpenCL	Embedded/Heterogeneous	[13]
Mc2For	2013-2014	Fortran95	CPU	[15]
eVariX/COLD	2013-2018	C	CPU/Heterogeneous	[57]
Mix10	2014	X10	CPU	[16]
m2cpp	2015-2018	C++	CPU	[58, 59]
StencilPaC	2016	C/OpenMP/MPI/OpenACC	CPU/GPGPU	[60]
MatJuice	2016	JavaScript	CPU	[61]
Latifis et al.	2017	C/SIMD	Embedded/System-on-Chip (SoC)	[62]

several information about variables in programs, such as their type, shape, size of each dimensions, and their structural properties, e.g. if a matrix is diagonal or triangular. The structural analysis is used to generate specialised arithmetic operations which work especially well on matrices with particular properties [11]. Furthermore, the analyses can be static (working from the source code) or dynamic (deferred to the runtime). For the analyses, FALCON represents MATLAB programs in the Static Single Assignment (SSA) form.

Moreover, FALCON is capable of transforming MATLAB programs to achieve better performance. However, the code transformation is manual and requires inserting annotations into the code. Further, the annotated code is transformed accordingly to one of the patterns in a database of rewriting rules. The rules include *algebraic restructuring* such as reordering matrices during multiplication, and *primitive-set translation* like loop interchange [11].

Otter. Otter is a compiler which not only translates MATLAB into C, but it also parallelises the code using Message Passing Interface (MPI) [49] and delegates numerical computations to parallel implementation of the Linear Algebra PACKage (LAPACK) library called Scalable LAPACK (ScaLAPACK) [12]. The compiler uses Static Single Assignment (SSA) as intermediate representation of MATLAB programs.

MATCH. MATCH is a compiler targeting heterogeneous platforms such as Digital Signal Processing (DSP) chips and FPGA. The compiler has a very detailed parser for the MATLAB language which development was detailed in an extensive report [63, 64].

Mat2C. Joisha and Banerjee, after years of work on shape inference for MATLAB [17, 65], have created MATLAB to C compiler called **Mat2C**. The compiler uses **MAGICA**, their shape inference engine.

MATISSE. MATISSE is a compiler from MATLAB language to C and OpenCL [13, 66], focusing also on embedded systems [45]. MATISSE uses aspect-oriented programming language called LARA [67] to specify missing information required for the compilation. LARA allows to declare, e.g. data types and shapes, which are dynamic information in MATLAB and usually unspecified in the MATLAB source code.

m2cpp. **m2cpp**¹ is a MATLAB to C++ compiler which generates parallel code using OpenMP [68] or Intel TBB [69] libraries. Moreover, for the computations, **m2cpp** utilizes Armadillo library. The compiler uses external file with *meta-information* such as types and shapes of variables required for the compilation.

¹<https://github.com/jonathf/matlab2cpp>

2.1.2 Transformation of MATLAB code

Although, much less popular, the code transformation of MATLAB programs was investigated by researchers as well. The biggest disadvantage of this approach is that the final gain from optimisation is limited to the performance of the MATLAB environment.

Loop vectorisation

Still today, loop vectorisation is considered as a primary tool for increasing performance of MATLAB loops². Menon and Pingali [9] have implemented automatic loop vectorisation in FALCON [11]. Moreover, they have performed one of the first studies on the effectiveness of loop vectorisation. Their results have shown that vectorisation is always beneficial and can be applied systematically.

Two further works on automatic loop vectorisation are mainly concern about shape analysis and the validity of the vectorisation. Birkbeck et al. [21] introduced the notion of dimensionality abstraction which helps to check if the code after vectorisation is correct. If not, their vectoriser can transpose arrays and check if this transformation made the code correct. Moreover, their vectoriser includes a database of common patten used in loop vectorisation.

The next study by Chen et al. [1] improved on the idea of dimensionality abstraction by introducing a data-flow analysis called *promoted shape*. The analysis was implemented in their vectoriser *Mc2Mce*³ along with automatic shape inference built on top of the Tamer [1, 70]. With the analysis, authors were able to perform loop vectorisation, but also vectorisation of the whole user-defined functions (also called *procedure vectorisation* [20]). Both presented approaches uses the same vectorisation algorithm by Allen and Kennedy [71, 72] as their base. However, both Chen et al. [1] and Birkbeck et al. [21] are not considering when vectorisation might not be beneficial, applying it only systematically.

Arithmetic simplification

Often, mathematical equations can be simplified and optimised. The same is true for MATLAB code which expresses these equations. In FALCON compiler, using annotations, it is possible to perform *algebraic restructuring*, such as reordering of matrices during multiplication, in order to reduce the number of performed arithmetic operations [11].

Menon and Pingali [73] introduced a whole framework for restructuring expressions called Abstract Matrix Form (AMF). AMF is an algebraic language for expressing semantics and transformations of element-wise matrix

²https://www.mathworks.com/help/matlab/matlab_prog/vectorization.html

³<https://github.com/Sable/Mc2Mc>

operations. Moreover, the framework contains several rewriting rules for simplifying computations which result in significant performance gain [9, 73].

Partial evaluation

Elphick et al. [74] presented a partial evaluation system for MATLAB where rarely (or never) changing parts of the programs are evaluated before the actual execution. In other words, partial evaluation is about evaluating static instructions and leaving dynamic instructions for later. The presented results were optimistic, however, their system considered only a small subset of MATLAB language.

2.1.3 Alternative execution environments

During the efforts to increase MATLAB performance, there were several attempts to build an alternative to the MATLAB environment.

MaJIC. Almási and Padua [18, 23] have built a Just-In-Time compiler for MATLAB using the FALCON infrastructure [11]. The compiler performs dynamic type, shape, and range analyses right before the code execution. Moreover, **MaJIC** reduces the amount of required temporary variables, automatically preallocates them, unrolls vector operations on small arrays, and performs preallocation for arrays which change their size during execution.

McMV. Chevalier-Boisvert et al. [22] have presented a new virtual machine (VM) for MATLAB with a JIT compiler. The JIT compiler is built using LLVM infrastructure [75], but it also has a mark-and-Sweep garbage collector and integration with numerical libraries: ATLAS, BLAS, LAPACK. Furthermore, the JIT compiler performs on-the-fly specialisation and optimisation of functions based on the type of their arguments. The compiled functions are later versioned and kept in a database for further function calls.

MATCH Virtual Machine (MVM). Haldar et al. [76], using an infrastructure of the MATCH compiler, have built a virtual machine for MATLAB language. The machine is capable of generating parallel code at runtime, thus, solving problems of having insufficient information about types and shapes of variables. Moreover, **MVM** executes parallel tasks in an out-of-order model to better exploit available resources.

2.1.4 Analysis of MATLAB programs

The analysis of MATLAB programs can be divided into three main categories: kind, shape, and type analysis. While the shape and type analyses are common for other languages and environments, kind analysis is unique to

MATLAB and results from two issues: (1) dynamic nature of the language where the same label can point to different objects in the course of the program execution; and (2) the ambiguity in the language syntax where the same expression `identifier()` could point to a function call or an array slice, two vastly different entities.

In our work, we use only a very simplified kind analysis based on the database of built-in functions, and *dimensionality* based shape analysis proposed by [21]. Moreover, when comes to types, we consider only `double` type representing double-precision floating point numbers.

Kind analysis

The goal of kind analysis is to statically determine if a `label` points to: a function call, variable or a package name. During the program execution, kind analysis is easily resolved with `exist label` function which not only tests the existence of the `label` in the workspace, but it also returns the `label` type (e.g. variable, function, MEX-file, class). However, the source code of MATLAB program obviously lacks this information, especially because the content of the `label` can dynamically change during the program execution.

The name *kind analysis* for this problem, along with a detailed semantics of how MATLAB 7 (year 2004) resolves this problem was described by Doherty et al. [77]. Moreover, Doherty et al. have created an algorithm to perform kind analysis using, e.g. flow-sensitive information. While previous works have performed kind analysis as well, most notably in FALCON [48], Menhir [19, 50], and MATCH [63, 64], they did not call it this way. Usually, their approach was to mark an identifier as a function if the identifier was in a database of built-in functions or when an inter-procedural analysis of user-defined functions has found a corresponding function definition. Otherwise, depending on the reaching definition, the label was deemed as a variable.

Shape analysis

In the world of array programming languages, shape analysis concerns with the size of multi-dimensional arrays, their rank (number of dimensions), and how their size changes after performing operations on them. As opposed to shape analysis in imperative languages like C, where compilers analyse properties of heap-allocated data structures, e.g. linked lists [78].

Lattice-based approach. The most common shape analysis for MATLAB, used in e.g. FALCON [48], Menhir [19], or McJIC [18], is based on a lattice defining a hierarchy of shapes (e.g. row vector, matrix, unknown) with rules for resolving shape when arrays are used as arguments to MATLAB functions. However, these analyses were always limited to matrices and, in many cases, they return an `unknown` shape, due to a lack of required shape information.

Algebraic approach. A completely other algebraic approach was presented by Joisha and Banerjee [17]. Their methodology expresses the shape of each variable as an algebraic expression. If a dimension or array rank is unknown, it is represented as a free variable in the equation. Therefore, the method instead of returning an `unknown` value, always returns an equation expression of the shape, even when parametrised with variables. The method with a dedicated inference engine called MAGICA [79] was implemented in Mathematica [80, 81]. The method is powerful, however, it is not only complicated, but also not very useful in many contexts, such as static code generation because the parametrised formulas with missing values are unacceptable for compilation, and they need to be resolved during the code generation phase. Nevertheless, the analysis was used in compiler `Mat2C` [54], built by the authors of MAGICA [79].

Dimensionality analysis. Introduced by Birkbeck et al. [21], *dimensionality analysis* is an example of specialised shape analysis performed only in the context of automatic loop vectorisation. This analysis has two goals: (1) to validate if the vectorisation was correct, and (2) to transpose arrays so that their shapes are compatible. Therefore, the analysis can be simple and not concerned about the exact size for each dimension. Dimensionality of arrays is represented using only three values 1, $*$ and r_i to indicate the dimension of size equals to 1, more than 1, or having a value obtained after a loop was vectorised along the i -th dimension, respectively. For example, this system represents a scalar as (1, 1), a row vector as (1, $*$), a column vector as ($*$, 1), and a matrix ($*$, $*$). However, this shape information must be manually inserted using directives and could not be inferred automatically.

Promoted shape. Demonstrated by Chen et al. [1], *promoted shape* is also a dedicated analysis to loop vectorisation. However, the analysis extends the concept of *dimensionality analysis* [21] to user-defined functions, and using data-flow techniques, to flow-sensitive code. Moreover, the implementation of the analysis inside the vectoriser `Mc2Mc` uses value analysis from Tamer [70], which can statically infer the exact shape of an array, if only possible. Therefore, in many cases there is no need for explicit directives with shape information like in the *dimensionality analysis* by Birkbeck et al. [21].

Type analysis

Some compilers had no type inference, instead, they incorporate a set of directives for specifying the types, shapes, and other information. For example in the work of Ramaswamy et al. [82], the presented compiler accepts directives e.g. `%! local float foo(128,128);` for type and shape. Other compilers have simple algebraic or lattice-based type inference like the work by Latifis et al. [62]. Similarly, FALCON implements a SSA-based forward propagation

inference algorithm [11, 47, 48]. In FALCON, if a type inference yield an unknown type, then the compiler generates code for dynamic type inference which is performed at runtime.

Domain-Specific Language (DSL) for type propagation. Type analysis is not only about inference rules because, with a vast number of built-in functions, it is hard to track and collect type information for them. For this problem, Dubrau and Hendren [70] have prepared a domain specific language for expressing how types propagate and change from the arguments of a built-in function to the result of the function.

2.2 Performance analysis

Precise performance analysis of computer system, at current state, is extremely complex which Abel and Reineke have recently summarised [83]:

“Modern microarchitectures are some of the world’s most complex man-made systems. As a consequence, it is increasingly difficult to predict, explain, let alone optimize the performance of software running on such microarchitectures. As a basis for performance predictions and optimizations, we would need faithful models of their behaviour, which are, unfortunately, seldom available.”

In this section, we describe several means to analyse and express performance of applications, without having an accurate model of the processor.

2.2.1 Metrics and models

When faced with hard problems, researchers delve into abstractions and simplifications. One example of such simplifications is performance metrics which describes one (or several) aspect of the program execution. Common metrics include: instructions per cycles (IPC) [84], loop balance [85], data reuse [86], etc. Metrics can be divided into dynamic and static, the former come as a result of measurements, while the latter are built just from analysing the program source code. Nevertheless, these metrics describe only small parts of the program execution like cache memory system, or utilisation of execution units in processor.

To get a more global view, we can use models such as polyhedral model [26], Roofline Model [32, 87–89], or Execution-Cache-Memory (ECM) [90]. These models are specialised in expressing how well the program will execute or what is the execution bottleneck. Nevertheless, each of them is only a part of the story, e.g. Roofline Model focuses on memory bandwidth (or caches in other available extensions); polyhedral model focuses on the order of memory access and computations.

Researchers also have created metrics dedicated to virtual machines, e.g. for JVM, however, these metrics mainly consider properties of the bytecode, and not the virtual machine directly. Consequently, none of these metrics and models are useful for MATLAB because of three reasons: (1) their target is the machine; (2) MATLAB is a runtime environment which needs to be modelled on its own; and (3) MATLAB is a closed source tool.

2.2.2 Hardware performance counters

Different approaches to performance analysis are based on pure measurements using for example hardware performance counters. These counters are special purpose registers on processors which collect information about program execution in form of performance events, e.g. amount of cache misses, processor stalls [91]. Although, at first, the set of performance counters was ever changing and unstable, recently the available performance events stabilised with the introduction of architectural events which are available in the whole line of the microarchitectures. Accordingly to Stéphane Eranian [92], the Intel's attitude towards performance counters has changed with the Intel Itanium processors. The second change can be observed with the introduction of Top-Down Microarchitecture Analysis (TMA) by Yasin [33]. TMA allows to find bottlenecks during program execution and to pinpoint them to a specific part of processor pipeline like the frontend, back-end, or execution units. Nowadays, performance counters are used in performance and power analyses, adaptive optimisations and many others.

Collecting performance counters. The popularity of performance counters is backed by the number of tools and libraries allowing easy access to them. A few solutions include: OProfile [93], perfmon2 [94], PAPI [43, 95], Tiptop [96–98]. Moreover, some applications for performance analysis allow to collect performance counters as well. Intel® VTune [99] collects performance events to perform the Top-Down Microarchitecture Analysis (TMA). However, the measurements come only from one run of the application and they are multiplexed. Therefore, TMA analysis is not usable for short programs or programs with time-varying behaviour.

In our work, we use PAPI library which already has an interface to interact with MATLAB. However, we only reuse the interface and extend it with new functionalities collected inside `mPAPI` (see Appendix B).

Accuracy of performance counters There is still an ongoing debate about the precision and accuracy of performance counters which might differ depending on the microarchitecture, implementation of the code responsible for accessing performance counters in the kernel, acquisition methods and libraries, among others. Weaver et al. performed several studies on the topic [100, 101]. General finding states there is usually a subset of accurate

and deterministic counters. However, the tests need to be performed for each new machine and its configuration [102]. In our work, we do not analyse the precision of performance counters, and instead, we only acknowledge their inherent imprecision for solving our problem of understanding the behaviour of MATLAB programs execution.

Feedback-directed optimisations. Schneider et al. [103] used performance events about cache activity to guide the Just-In-Time (JIT) compilation process. This type of optimisations is known as Feedback-Directed Optimisations (FDO); also called Profile-Guided Optimisations (PGO), because the Just-In-Time compiler monitors programs and collects performance events from their execution. Later, accordingly to the feedback from their execution, a decision about future transformation is taken.

Characterising programs with performance counters. Stéphane Eranian [92] analyses how performance counters can be used to understand performance of memory subsystem on processors. In his work, he points out that performance counters are important for program analysis because they are more common nowadays, do not require program recompilation, and have a small overhead (especially in comparison to, e.g. simulation).

The study by Eeckhout et al. [104] used hardware performance counters to analyse interactions between various components like Java Virtual Machines (JVM), processors, and programs. The results show that differences between JVMs implementations are greater than the difference from running various benchmarks on the same implementation. Similarly to our study, Eeckhout et al. looked directly into the performance events to see how interpreted programs perform on processors.

The study by Sweeney et al. [105] described a methodology to analyse performance of Jikes RVM (Research Virtual Machine) using traces with performance events. The traces allow to better understand the interactions between various components of Java program execution: the application, virtual machine, operating system, and the microarchitecture. However, authors noted that in their case, traces of performance counters are not enough to explain certain performance phenomena.

Machine Learning with performance counters. In their seminal work, Cavazos et al. [34] pioneered the use of hardware performance counters as a representation of code examples for machine learning. In this work, the training data set consists of programs before code transformations. Each program is described with values of performance counters collected from their execution. The training set is then used to create a model which selects the best transformation.

2.2.3 Profiling

Presented models and use-cases of hardware performance counters assume that programs execute in a regular manner. However, as in the case of MATLAB or programs with control dependences, it is not the case. In this subsection, we show two approaches to the analysis of time-varying and phase behaviours.

Time-varying behaviour. One of the first studies of program properties changing throughout the program execution was a study of large-scale patterns in SPEC95 benchmark suite by Sherwood and Calder [106]. The work looked for patterns in performance profiles of e.g. instructions per cycles (IPC), cache miss rate or branch prediction miss rate, in terms of committed instructions. An interesting result of the study was finding *cyclic behaviours* in benchmarks, which can also indicate for how long we should run the benchmarks to obtain representative results. Subsequent studies by Sherwood et al. improved the analysis of time-varying and large-scale patterns by either creating an automatic machine-independent technique for finding large-scale pattern [107] or proposing a hardware (and software) tracking and prediction method for reoccurring phase behaviours [108]. Although our work deals with small scale time-varying changes and lacks of phase behaviour, the work of Sherwood et al. is an early example of a detailed analysis using performance profiles. In another interesting study, Duesterwald et al. [109] argued that time-varying behaviours are important and should be incorporated into adaptive systems to improve program performance and energy consumption. The results show that programs have time-varying behaviours at even small scales which could be used e.g. to predict the value of one metrics based on another (*cross-metrics*).

Vertical profiling. Across three papers, Hauswirth et al. [110–112] explored the idea of vertical profiling, a methodology for understanding and correlating performance data obtained over time from multiple levels of abstractions: server, hardware, virtual machine (VM), operating system (OS), application. At the core, in correlating performance from multiple levels of abstraction lies the same idea as in our performance event profiles from Chapter 3. However, in our case, we correlate multiple performance events coming from the same abstraction – a processor. In the second paper, authors evaluated several techniques for automating trace alignment coming from different measurements [111]. So far, in our work, we have used only a small amount of performance events which are measured at once.

Warm-up state. A particular case of time-varying behaviour is a warm-up state which occurs for virtual machines and Just-In-Time (JIT) compilers

when a class or resource is loaded for the first time, or during the profile-guided optimisation phase (when JIT compiler monitors how programs execute).

In a 3 years long study, Barrett et al. [36] have analysed warm-up phases of popular interpreters and virtual machines, e.g. JavaScript V8, Python PyPy, Java HotSpot. The analysis was based on the repeated execution of the same program in order to obtain a performance profile of its execution times. The results of the work show that virtual machines and JIT compilers often have inconsistent warm-up states. Moreover, for some cases, that JIT compilation decreases the performance of programs.

2.2.4 Performance of execution environments

Compilers and compiled programs are not the only target of performance analysis. For many years, researchers have been working on the performance analysis of interpreters, virtual machines, and Just-In-Time compilers (some of these examples, we have already mentioned in previous sections).

Branches and jumps. Since their creation, interpreters were considered being slow because they create an additional layer of abstraction between the program and the hardware. Moreover, by definition, they interpret instructions one by one, thus making impossible optimisations which work on two or more instructions. At the time, slow branches and jumps were often considered as the root cause of performance problems [6, 113–115].

The notion of branch misprediction as the main problem was widespread for many years. Until the study by Rohou et al. [116], where researchers analysed again this concept and compared current and new techniques for branch prediction. The results showed that the new microarchitectures have improved to a level, that the branch misprediction was no longer a problem.

Decomposing performance. Several works have tried to decompose performance of various interpreted languages. Barany [117] has tried to analyse the performance of particular features in the CPython interpreter. His approach was to modify the interpreter so it is possible to disable a single feature and perform tests with and without it. However, this approach is not applicable to MATLAB because it requires an access to the source code of the execution environment.

Carchiolo et al. [118] have tried to analyse the activity of Python dynamic features, e.g. reflection, dynamic typing, using performance profiles. Their findings show that the highest activity of these features occur mostly during the program start-up.

2.3 Conclusion

From the analysis of related work, we have learned several important points:

- The majority of research about accelerating MATLAB programs is dedicated to compiling MATLAB to other languages, mainly C and Fortran [45, 58, 62]. In our work, we focus only on code transformations and the performance analysis of the MATLAB environment.
- There were no prior work on execution models for the MATLAB environment. In Chapter 4, we introduce an execution model, *instruction tree*, for MATLAB.
- Recent research on vectorisation of MATLAB loops shows that the execution model and the deep understanding of MATLAB runtime are crucial for successful application of vectorisation [1]. In Section 5.4, we apply profile-guided loop vectorisation to MATLAB programs.
- Performance profiles and traces were mostly used to find bottlenecks in applications, analyse time-varying and large-scale behaviours, or to correlate performance of multiple components. In Chapters 3 and 4, we use performance profiles to discover the type and order of JIT-compiled instructions coming from MATLAB expressions.
- Not every performance analysis tool works well with execution environments such as MATLAB. Therefore, we have decided to implement our own tool `mPAPI` (see Appendix B) for accessing hardware performance counters directly from MATLAB code.
- The performance analysis of interpreters, virtual machines, and Just-In-Time (JIT) compilers is focused on open source projects; thus, it cannot be applied to MATLAB. Instead, in our work we focus entirely on the program performance observed from the processor perspective using performance event profiles (PEP) from Chapter 3.

Chapter 3

Performance event profiles

Résumé

Afin d'analyser le fonctionnement d'un programme dans un environnement propriétaire tel que MATLAB, nous utilisons des profils d'événements de performance. Ces profils décrivent l'exécution du programme sur l'unité centrale directement, en capturant également leur comportement dans le temps. Par conséquent, ils contournent les composants responsables de l'exécution qui sont difficiles à analyser, comme le système d'exploitation (OS) et l'environnement MATLAB lui-même. Grâce à ces profils, nous pouvons mesurer les erreurs de cache, les erreurs de prédiction de branchements, les blocages dans le pipeline et de nombreuses autres caractéristiques d'exécution.

Avec les profils d'événements de performance, nous pouvons décomposer l'exécution des programmes en sous-composantes de calculs appelés régions d'exécution. Ensuite, nous pouvons analyser chaque région séparément. De plus, nous pouvons observer le résultat de la compilation en temps réel (JIT) en recherchant les régions composées de plusieurs opérations (elles correspondent à un seul bloc de base dans le code machine). Ce faisant, nous obtenons une image complète de la compilation JIT des expressions MATLAB.

La mesure à l'aide de profils d'événements de performance exige de notre part la mise en place de plusieurs paramètres. Le premier est le seuil d'échantillonnage qui indique la fréquence de lecture de la mesure des compteurs de performance. En raison de l'effet d'observation, un échantillonnage dense pourrait affecter considérablement l'exécution mesurée. En même temps, si l'échantillonnage est faible, nous pourrions perdre trop d'informations. Un autre critère à considérer est le type d'événements de performance. Avec la grande quantité d'événements (171 événements sur l'architecture Skylake), il est nécessaire d'examiner attentivement leur signification.

Introduction

In this chapter, we unravel how profiling of performance events allows to accurately describe program behaviour. Still today, program behaviours are described mostly with one dimensional metrics such as loop balance [85],

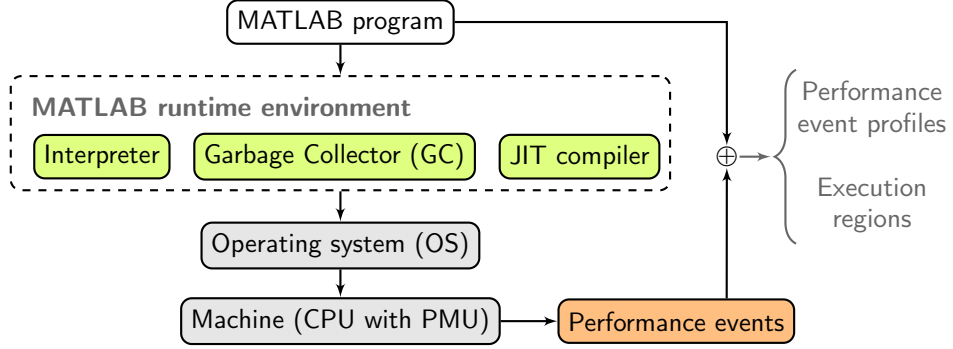


Figure 3.1: A simplified view on components taking part in the execution of a MATLAB program.

cycles per instructions (CPI) [84], arithmetic intensity and memory bandwidth from the Roofline Model [32], and many more. However, those metrics fail to encompass non-linear and time-dependent behaviour of many programs, and they are a not sufficient help in exploring the runtime semantics of MATLAB programs.

In Section 3.2, we present the case of MATLAB program for which standard metrics fail to capture the behaviour; Section 3.3 describes the creation of performance profiles; Section 3.4 shows a methodology for the classification of segments with interesting properties in profiles; and finally Section 3.5 presents one case study for performance profiles used to characterise the cost of indexing arrays in MATLAB.

3.1 Overview

Code transformation is a common way to improve program execution where a code is rewritten to obtain better performance. However, the knowledge of which part of the code is concerned and how to transform it requires precise information about the underlying algorithm, the programming language, the used compiler or interpreter, the runtime system, the operating system, the processor and its microarchitecture (see Figure 3.1).

In an ideal world, each of the components mentioned above is well-documented and provided as open source. Unfortunately, MATLAB is a proprietary, closed source computing environment. Therefore, MATLAB programmers have no precise technical information about the interpreter, Just-In-Time (JIT) compiler, garbage collector, nor the runtime system. Reverse engineering of MATLAB seems too difficult due to the JIT compiler which dynamically creates new machine code. Moreover, the dynamic binary instrumentation with tools, e.g. DynamoRIO [119] or Intel PIN [120] is inherently complex because of the static machine code of the interpreter

mixed with the dynamic code generated by the JIT compiler.

MATLAB environment runs on an operating system (OS) responsible for running it and other processes, handling I/O, scheduling work between processors, communicating with external devices among others. The environment and the OS execute on processors with unique microarchitectures. Processors are one of the most complex human-made systems, also not well-documented [83]. Nevertheless, in the inherent mist of complexity of the program execution, processors have a performance monitoring unit (PMU) capable of measuring and recording the behaviour of program execution. Figure 3.1 presents these common components taking part in the execution of MATLAB programs.

Surprisingly, a solution to discover the behaviour of the MATLAB environment is to forget entirely about the inner-workings of MATLAB. Instead, we focus on the link between an input MATLAB program and its observed runtime behaviour in the form of performance events as shown in Figure 3.1 (arrows and the \oplus symbol). With the use of the PMU, it is possible to run the program and to simultaneously record its execution on the processor. In this chapter, we focus on exploring methods to measure and record processor behaviours and to analyse the results to create the link between a given program and its execution.

3.2 Motivation

Observing processor behaviour using performance events during program executions shows how the processor performs MATLAB programs. Performance events indicate two critical things: (1) what kind of instructions the program issues on the processor and; (2) how well the program executes on the processor. Group (1) contains performance events which count, e.g. the number of *retired instructions*¹. The number of retired instructions coupled with instruction type and their latency in cycles, could give an excellent performance prediction on sequential processors [121]. However, multi-core processors are vastly more complex; they have multiple cores; they perform many instructions at once and in an out-of-order fashion, and they can speculate and execute instructions ahead-of-time. Thus, group (2) of performance events fills this gap by describing how well components and resources on a processor perform at almost every point of the processor pipeline.

Nevertheless, the performance events obtained after the program has finished its execution indicate only the total number of executed instructions

¹*retired instructions* — executed instructions from the correct execution path or floating-point operations. Modern processors *speculate* about which execution path might be taken in the near future and they execute instructions from such a path to better utilise available resources.

²*array slicing* — extraction of a subset of array elements. Some examples from MATLAB: `A(1:N)` extracts elements from 1 to N, `B(1:2:N)` returns only odd elements.

Performance profiles for Schönauer Vector Triad (striad)

Run on machine M1; MATLAB=R2018b; threads=1; sampling threshold=100000; N=1000000.

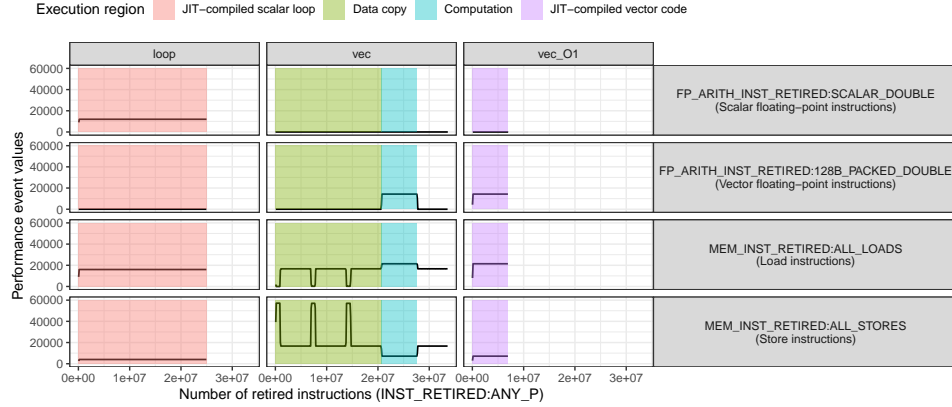


Figure 3.2: Performance profiles for the *striad* kernel in three versions as a: (1) loop computation; (2) vectorised instructions on all data; (3) vectorised operations with array slicing². The figure depicts four interesting regions: **JIT-compiled scalar loop**, **data copy**, **computation**, and **JIT-compiled vector code**.

and potential performance bottlenecks for the whole program (with the use of, e.g. Top-Down Micro-architectural Analysis [33]). The missing information in the full description of MATLAB execution is the time when the instructions execute (e.g. *do floating-point operations execute at the end of the program?*) also, how well at each stage of the program those instructions perform (e.g. *do cache misses occur while executing floating-point operations?*). Therefore, the solution is to gather performance events over the program execution time, where “time” actually means another performance event such as *retired instructions* or *total cycles*. The result in Figure 3.2 shows *performance profiles* indicating not only the amount, but also when performance events happen after some amount of *retired instructions*.

Listing 3.1: Three versions of the same kernel the Schönauer Vector Triad *striad*. In MATLAB, it is common to express loops as array functions (vectorised code).

```

1 % Loop computation
2 for ii = 1:N
3     a(ii) = b(ii) + c(ii) .* d(ii);
4 end
5 % Vectorised with explicit indexing
6 a(1:N) = b(1:N) + c(1:N) .* d(1:N);
7 % Vectorised over whole vectors
8 a = b + c .* d;
```

Combining multiple *performance profiles* helps us finding *execution regions*

which represent distinctive stages of the computation. Possible regions include e.g. (1) *data copy* where we observe proportional amount of memory loads and stores; (2) *computation* with a high amount of floating-point operations; (3) *scalar loop* with a constant amount of each performance event; or finally (4) *JIT vector code* which is similar to *scalar loop*, except for the occurrence of packed floating-point vector operations.

Figure 3.2 presents these four execution regions for 3 versions of the Schönauer Vector Triad kernel (*striad*) listed on Listing 3.1. Execution regions can also be defined using the notion of how well a given part of the program runs, e.g. by indicating memory or compute bound regions [33]. In Figure 3.2, we observe that the `loop` and `vec_01` have regular profiles, because both compile to one *instruction block*, the concept, which we will explore in Chapter 4 for the case of vectorised code.

The `vec` code version in Figure 3.2 is a vectorised loop with explicit array slicing which generates data copy, as opposed to `vec_01` version without array slicing. The main difference between these two codes is the data copy region marked in green (and one unmarked *store region* at the end of the `vec` code). Without performance event profiles (PEP) and just by using performance counters in the counting mode, it is not possible to uncover these execution regions, their type, order, and performance. Finally, just by looking at Figure 3.2 it is not a surprise that the fastest code is `vec_01`.

In the next part of the chapter, we investigate how to build, analyse, and use *performance event profiles* (PEP) with *execution regions* in order to explore and understand the behaviour of MATLAB programs.

3.3 Building performance profiles

In this section, we define *performance event profile* (PEP) and *execution region* formally. Moreover, we show how to build performance profiles using our `mPAPI` tool and the `PAPI` library [95].

Performance profile describes changes of performance event measured over time. Fortunately, performance counters on modern CPUs work in two modes [122]: (1) *counting mode*, where the values of a performance event are accumulated in a single register; and more importantly (2) *sampling mode*, where measurements are taken in intervals equal to a specified *sampling threshold* of a performance event used as the “time” (e.g. on every 1 000 000 retired instructions). Therefore, the capabilities of building performance profiles are already inside modern CPUs with mechanisms of *counter overflow*, Event-Based Sampling (EBS), and recently Precise Event-Based Sampling (PEBS) on Intel processors [91, 123].

Performance profile. We define a performance profile as $P = (T, M)$ where $T = (t_1, t_2, \dots, t_l)$ and $M = (m_1, m_2, \dots, m_l)$ are sequences of sam-

pling times and measurements $t_i, m_i \in \mathbb{N}$, moreover, the sampling time is strictly increasing $\forall i < j : t_i < t_j$. Both T and M have the same length l . We say the length of the profile $|P| = l$. Moreover, T and M have their own *event domain* \mathcal{E} , which states what performance event the sequence represents. The universe of event domain \mathcal{E} contains any performance event available on the CPU. For example, a sequence of *level 1 cache misses* measurements M has the event domain as follows $\mathcal{E}(M) = \text{L1D:REPLACEMENT}$. Along with sequences T and M , the event domain of the whole performance profile $\mathcal{E}(P) = (\mathcal{E}(T), \mathcal{E}(M))$ creates a full description of the profile. Event domains specify the profile content, where M holds the measurements and T holds the sampling time with embedded sampling intervals between every pair of measurements.

Function $\Delta(X, i) = x_i - x_{i-1} : x_i, x_{i-1} \in X$ calculates the difference between elements i and $i - 1$ of sequence X (however, for $i = 1$ it is always $\Delta(X, 1) = x_1$). In an ideal case when measurements are precisely taken, Equation (3.1) about the sampling time T always holds, which means that observations are obtained in equal sampling intervals. However, this is rarely the case due to measurement errors.

$$\exists \hat{t} \in \mathbb{N}, \forall i \in [1, l] : \Delta(T, i) = \hat{t} \quad (3.1)$$

The constant \hat{t} from Equation (3.1) expresses only a theoretical sampling threshold. Therefore, in an ideal world, single measurement V in the *counting* mode of a performance event is equal, or at least very close, to the sum of measurements $m_i \in M$ of the same event obtained in the *sampling* mode: $V \approx \sum M$.

This correspondence of the counting and sampling modes indicates an interesting property of performance profiles, that the value of sampling threshold affects values of measurements in the profiles (this can be seen in Figure 3.4 from Section 3.3.2). When hardware performance counters collect occurrences of performance events, they never stop, no matter if we sample the counters more or less often. Thus, if we collect performance counters more frequently, the value of counters increase only a little, than when we sample them rarely and give them a time to grow. Taken to the extreme, a very rare, one-time sampling is equivalent to measuring performance counters in the counting mode. In other words, performance event profiles are nothing like the result of sampling a physical signal.

Profile group. In fact, building only one performance profile at a time is usually poor resource management, because the rest of *hardware performance registers* are not used (if available). Therefore, it is beneficial to measure several performance profiles at once, creating a *group* $G = \{P_1, P_2, \dots, P_g\}$ of g profiles. In a system with N hardware performance registers, we can create only $g = N - 1$ profiles, because the last register measures the

sampling event. Moreover, profiles in a group are *aligned*; thus, we can simplify the group definition to $G = (T, \{M_1, M_2, \dots, M_g\})$ where the profiles and their measurements share the sampling time T . As in the case of performance profiles, groups have their event domain defined as follows $\mathcal{E}(G) = (\mathcal{E}(T), \{\mathcal{E}(M_1), \mathcal{E}(M_2), \dots, \mathcal{E}(M_g)\})$.

3.3.1 Selecting the sampling event

When a processor executes a program, it generates various performance events describing the program behaviour. Those events are measured with programmable hardware performance counters. The simplest example of a performance event is the number of processor cycles which is counted each time the processor moves forward the program execution. However, other performance events also increase their values as the execution progress. Therefore, in principle, any performance event can be the sampling event which expresses the progress of program execution and as such, can be used as the base of performance profiles.

Two popular choices for the sampling event on Skylake architecture are the number of retired instructions (`INST_RETIRE:ANY_P`) and the total amount of processors cycles (`CPU_CLK_UNHALTED:THREAD_P`) depicted in Figure 3.3. Both events increase, but in a slightly different manner. While processors cycles always change (otherwise the processor is not running), instructions not always retire on every cycle because processors sometimes experience pipeline stalls and none of the executed instructions are retired [84]. Therefore, some performance events have time gaps, when the processor is running (and the cycles increase), but the event is not changing its value. This leads to a conclusion that not every performance event is a good candidate for the sampling event which supposed to be like “time”, changing along the progressing program execution. Moreover, this is even more visible when some performance events are never meant to happen like vector floating-point operations in a scalar program.

The “time” gaps in program execution can manifest as sudden spikes of some activity like in the case of the number of requests to the external memory (`OFFCORE_REQUESTS:ALL_REQUESTS`) in Figure 3.3. Those two red spikes are seen on the performance profile of store instructions (`MEM_INST_RETIRE:ALL_STORES`) and what they really mean is that while nothing happened in terms of requests to the external memory (horizontal axis), a lot of store instructions were performed in caches without reaching to the RAM memory (vertical axis). In other words, although, those requests look like not the best candidate for the sampling event, they still bring new information and insights. Moreover, the requests to the external memory are an order of magnitude less frequent, thus, we use an order of magnitude smaller sampling threshold than for retired instructions and processor cycles. In general, performance events differ in their rate of occurrence.

Profiling with various sampling events

Schönauer Vector Triad kernel (striad) run on machine M1; MATLAB=R2018b, threads=1, N=1000000.

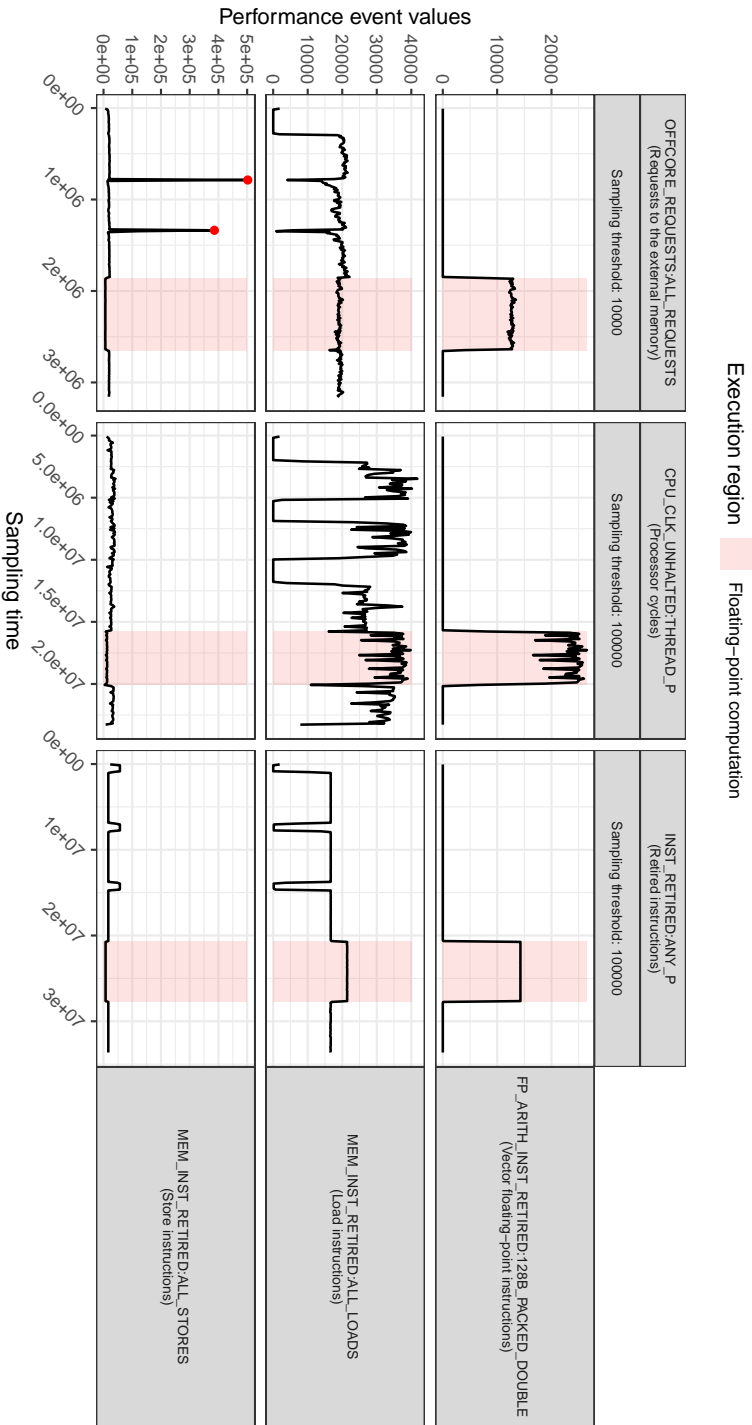


Figure 3.3: Performance profiles built using various sampling events. The choice of the sampling event affects the duration and properties of performance profiles.

What versus How the program executes. The sampling events, retired instructions and processor cycles, are much easier to reason about than, e.g. the number of requests to the external memory. The use of retired instructions (on Skylake and Coffee Lake architectures `INST_RETIRED:ANY_P`) in performance profiles help answering questions: (1) how many instructions of type X execute every n retired instructions? (2) how many cache misses do n retired instructions generate? or (3) what types of instructions do n retired instructions represent? Where n stands for the measured sampling threshold, as opposed to the theoretical \hat{t} threshold.

The number of retired instructions is the most used performance event in our study, because when used in performance profiles, it describes the intent of a computation (a signature of the computation). What we mean by that is the ability to answer question (3) and to find segments of performance profiles with particular properties, for example the floating-point computations like in segments highlighted in Figure 3.3.

Although, processor cycles (on Skylake and Coffee Lake architectures `CPU_CLK_UNHALTED:THREAD_P`) can help us answer the same questions as retired instructions, they do it in a slightly different way. Processor cycles are the closest to the real physical time as any performance event can be. Moreover, as we have mentioned before, the processor experiences pipeline stalls resulting in a noise and ever changing length of particular segments and whole profiles. Therefore, processor cycles as the sampling event can additionally show: (1) where programs experience stalls by prolonging execution of some parts of a program? and (2) for how long a part of a program executes? (a question explored in our case study Section 3.5). In conclusion, performance profiles built with retired instructions show what we want to execute; while profiles measured using processor cycles show how it executes.

3.3.2 Selecting the sampling threshold

When comes to the size of the sampling threshold, we face a multi-criteria dilemma because as in any sampling methodology, sampling too much overflows us with a stream of useless and repeated data, also creating huge files. On the other hand, sampling rarely can easily hide the existence of interesting execution properties like sudden spikes or valleys of the measured events. Therefore, in this section we look closely into how changing sampling threshold affects these criteria and performance event profiles (PEP).

Impact of the sampling threshold. Figure 3.4 presents how performance event profiles change with an increasing sampling threshold. Each performance profile represents an execution of the same kernel `striad` with the same data size $N = 1000000$. Apart from the obvious change of the level of details (the more we sample performance counters, the more detailed the profile is), Figure 3.4 presents a perfect comparison between the intent of

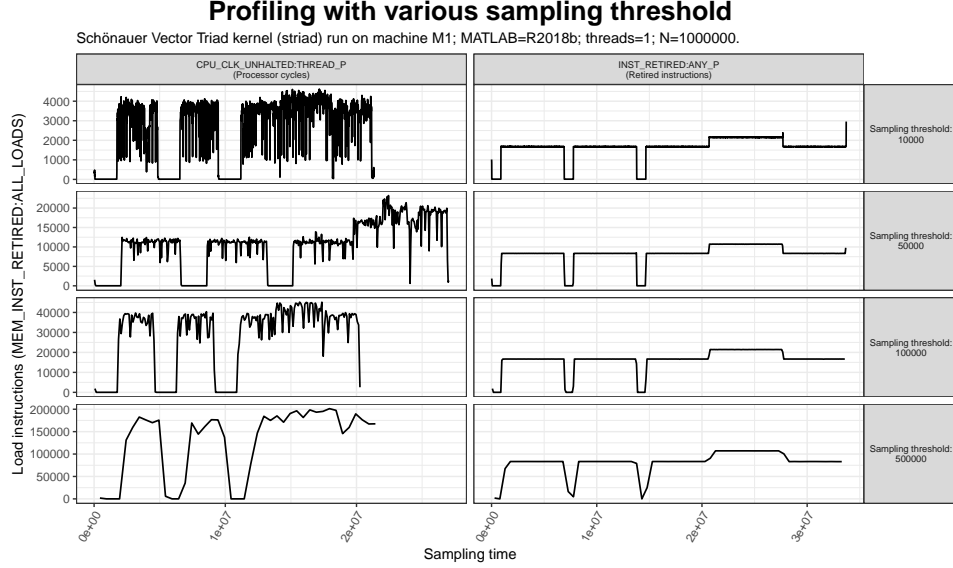


Figure 3.4: Performance profiles built using various sampling thresholds depicting load instructions (`MEM_INST_RETIRED:ALL_LOADS`). Profiles differ in their level of details, but also in their length.

a program and its real execution. Performance profiles built using retired instructions (`INST_RETIRED:ANY_P`) all have the same length and identically placed execution regions, because MATLAB executes a given code with the same set of instructions in the identical order. This is because MATLAB has a single-tier Just-In-Time (JIT) compiler [40] which, for a given code, always applies the same set of optimisations generating the same machine code. However, as seen in performance profiles built using processor cycles (`CPU_CLK_UNHALTED:THREAD_P`), the real execution on a machine has more noise and it is less predictable with unequally stretched and contracted parts of the computation, even when executing the same code many times.

The choice of the sampling threshold affects also the values of performance profiles. Figure 3.4 shows how increasing the sampling threshold from 10 000 to 50 000, increases 5 times the maximal amount of memory load instructions on profiles with retired instructions (`INST_RETIRED:ANY_P`). The reason is very simple, the performance profiles are not a sampled representation of a signal. Instead, they contain values sampled from always increasing performance counters. Therefore, without changing the rate of counting, if we sample the counters more often then the collected values are smaller than if we sample them less frequent.

Measurement impact. The theory of the observer effect in physics states that the observation of a phenomena changes the measurements [124]. Unfor-

Impact of the sampling threshold on performance profiles

Schönauer Vector Triad kernel (striad) run on machine M1; MATLAB=R2018b; threads=1; N=1000000.

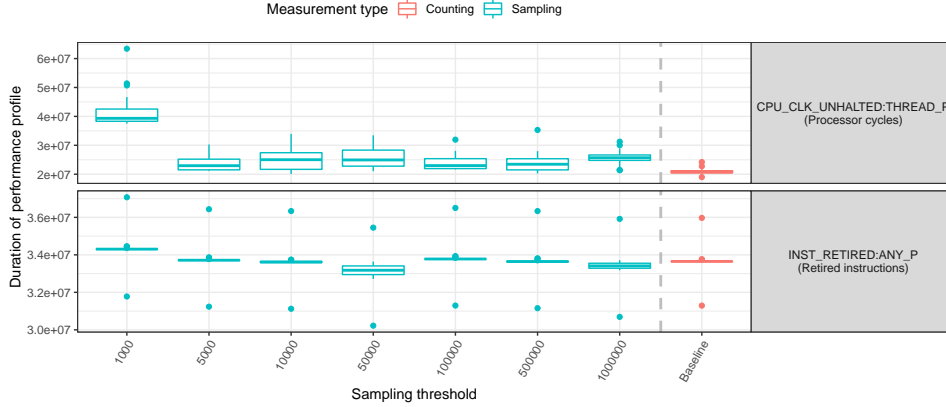


Figure 3.5: Impact of the sampling threshold on the duration of performance profiles. Frequent sampling of profiles, based on processor cycles, considerably affects the program execution.

Unfortunately, a similar effect is visible in Figure 3.5, where the act of sampling impacts the final measurements. This effect is severe for performance profiles built on top of processor cycles because of several sources of overhead: the hardware sampling mechanism, running `mPAPI` (introduced in the next Section 3.3) in MATLAB to gather measurements, and saving the measurements after each program execution in an external file. Therefore, the more frequent we sample performance counters (with smaller sampling thresholds), the longer it takes for the programs to finish. However, the impact of sampling is not considerable for performance profiles built using the number of retired instructions, because the mentioned overheads add only a small amount of instructions (e.g. the `mPAPI` code is short and simple).

The impact of the sampling mode is compared to a baseline counting mode in Figure 3.5. In the case of processor cycles based profiles, the difference is as high as 100 % for the sampling threshold of 1000. This clearly suggests that below a certain value of the sampling threshold, the collected measurements lose their precision significantly. However, before jumping to a conclusion about the exact limit of usable sampling thresholds, we need to remember that `mPAPI`, used in tests, implements only one measurement scheme. Therefore, we leave the exact estimation of this limit for future work along with testing other implementations of the measurement scheme in `mPAPI`. For now, we only make sure that the selected sampling threshold in our experiments has the impact on measurements lower than an arbitrary 20 %.

Size of profile files. The value of the sampling threshold impacts the number of collected measurements. Therefore, it affects the size of result files

Table 3.1: Size of profile files with changing sampling threshold. Each file contains 60 profiles of load instructions. The tests use the same benchmark `striad` with the same data size N equal to 1 000 000 (30 repeated profiles come from MATLAB R2018b and another 30 from R2015b).

Sampling threshold	CPU_CLK_UNHALTED:THREAD_P	INST_RETIRED:ANY_P
1000	76.2 MB	67.8 MB
5000	9.0 MB	13.4 MB
10 000	4.4 MB	6.7 MB
50 000	897.2 kB	1.3 MB
100 000	451.6 kB	675.7 kB
500 000	106.3 kB	143.9 kB
1 000 000	58.7 kB	77.5 kB

containing the values of performance event profiles (PEP). Table 3.1 presents how the size of profile files changes with changing sampling threshold. If we consider the high impact of frequent sampling depicted on Figure 3.5, we notice in the Table 3.1 that a 5-fold increase of the sampling threshold from 1000 to 5000 decreases the file size 8.47 times for processor cycles based profiles (middle column). On the other hand, for the retired instructions based profiles, the change of sampling threshold affects proportionally the size of result files, as expected.

3.3.3 Performance profiles with mPAPI

Apart from manually programming *hardware performance counters*, several libraries give easy access to both modes of measurement: *counting* and *sampling*. In our work, we have focused on the PAPI library [95] because this library is a comprehensive, open source, up-to-date, actively maintained solution with a C API which makes it possible to integrate with MATLAB through C MEX API ³.

For that reason, we have created `mPAPI`⁴, an open-source profiling tool for MATLAB. `mPAPI` supports both counting and sampling modes of collecting performance events. In this section, we focus only on the sampling mode, but for a detailed description of `mPAPI`, please refer to Appendix B.

In the sampling mode, `mPAPI` creates *performance traces* which store the raw information about the occurrence of the performance events. Before collecting values from performance counters, we need to define trace parameters. For this task we have one function `trace_register()`, which takes four arguments: (1) sampling event used as the sampling domain; (2) sampling threshold value which indicates the interval for reading performance counters;

³<https://www.mathworks.com/help/matlab/call-mex-files-1.html>

⁴<https://github.com/quepas/mPAPI>

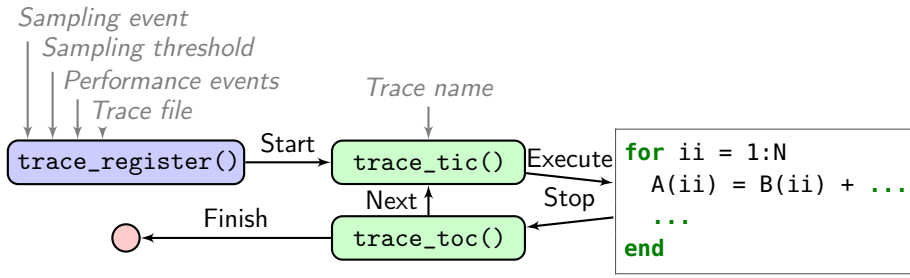


Figure 3.6: Workflow of creating performance traces with mPAPI.

(3) list of performance events to measure; and finally (4) path to the trace file which will store our measurements. The `trace_register()` function calls `PAPI_overflow()` of PAPI library which uses either a hardware event-based sampling (if available on the CPU) or a software timer which periodically checks the values of the performance counters, thus imitating the event-based sampling behaviour. Moreover, the `PAPI_overflow()` function allows tracking the Program Counter (PC) address at the time of the measurement. However, mPAPI does not store this information because it is unfeasible to map the PC address with a particular region of MATLAB code (for the reasons mentioned at the beginning of this chapter).

Next, after successful registration, we have two functions for starting and ending a performance trace. Function `trace_tic()` starts a new performance trace and takes an argument, *trace name*, which helps to distinguish between traces. Finally, function `trace_toc()` marks the end of the performance trace and stores the result. For a single registration, we can create many performance traces of various benchmarks and their repeated executions. Figure 3.6 presents the workflow of creating performance traces using mPAPI.

Listing 3.2: Performance trace over *retired instructions* (INST_RETIRED:ANY_P) with measurements of three performance events. The trace comes from the execution of the `striad:vec_01` benchmark on MATLAB R2018b.

```

1 @trace_start:R2018b:1:1:striad:vec_01:1000000:2
2 @perf_events:INST_RETIRED:ANY_P,FP_ARITH_INST_RETIRED:128
   B_PACKED_DOUBLE,MEM_INST_RETIRED:ALL_LOADS,MEM_INST_RETIRED:
   ALL_STORES
3 62593,2790,18189,9392
4 162592,17072,39626,16533
5 262518,31345,61027,23669
6 362593,45638,82351,30816
7 462593,59922,103787,37958
8 ...
9 6962583,988316,1494990,502154
10 7062518,1000000,1518534,511403
11 @trace_end
  
```

The *trace file* is a collection of many performance traces encoded as text (Listing 3.2 presents single performance trace). Keywords `@trace_start` and `@trace_end` indicate the start and the end of the trace respectively. The text after `@trace_start` is a *trace name* which usually describes the measured code. The third keyword `@perf_events` defines the event domains of the measured profile group $\mathcal{E}(G)$. Finally, each row of numbers contains one measurement for each performance event, the first number in a row is the sampling event.

Nature of hardware performance counters is to count and sum up values of performance events. Thus, measurements in columns from the trace file never decrease. Therefore, to obtain performance profiles, traces need to be pre-processed and requires calculating the difference between the current i -th row and the previous $(i - 1)$ -th row. After the calculation, it is easy to spot the possible inaccuracies of the sampling mechanism. In our example (Listing 3.2), the trace supposed to contain measurements taken on every 100 000 retired instructions, but the sampling time shows a bit different values $\Delta(T) = (62593, 99999, 99926, 100075, 100000, \dots, 99935)$. However, the inaccuracy is not severe enough to invalidate the conclusion coming from the following sections and chapters.

3.4 Finding execution regions

In this section, we explore a simple approach for finding and analysing regions with interesting execution properties in the performance profiles. The properties include the occurrence of specific processor instructions, (non)existence of performance bottlenecks, or simply, appearance of performance events with particular values.

Execution region. A section of the program execution with particular properties is an *execution region* expressed as a binary predicate $\varphi(t) : T \rightarrow \{0, 1\}$. The predicate marks an interesting region in the domain of the sampling event, thus, the predicate indicates when the region starts and ends during the program execution. We assume predicate φ is applied to measurements coming from the same profile group or from aligned groups. Otherwise, ambiguities appear, e.g. when two performance profiles P_1 and P_2 have different lengths $|P_1| \neq |P_2|$.

$$\begin{aligned} \varphi'(t) = & \text{MEM_INST_RETIRED:ALL_LOADS}(t) > 0 \wedge \\ & \text{MEM_INST_RETIRED:ALL_STORES}(t) > 0 \wedge \\ & \frac{\text{MEM_INST_RETIRED:ALL_LOADS}(t)}{\text{MEM_INST_RETIRED:ALL_STORES}(t)} \approx 1 \end{aligned} \quad (3.2)$$

Consider the problem of quantifying a cost of data copy in programs.

We could define data copy as a region presented in Equation (3.2). In the definition, the data copy has a non-zero amount of load and store instructions, and the ratio between loads and stores is close to 1. We have chosen here to count loads and stores, because during data copy each loaded element should be stored in a new location. However, there are plenty of others, equally adequate, performance events capable of describing a desired execution region (which we will see in the task of detecting *instruction blocks* in Table 4.1 from Chapter 4). An example of a search for meaningful performance counters is the work of Molka et al. [125] where researchers have embarked on a quest of finding the best counters for characterising utilisation and performance of memory subsystem. Finally, the exact definition of an execution region always depends on its purpose as well as on availability of performance events on a given machine.

Having a defined predicate φ' , we apply it to a group of performance profiles which contains mentioned load and store events. The result is a region where the data copy occurs, or at least program execution which shows similar properties. From that region we extract its start and end time to compute the length of the computation (or the amount of data transfer), in terms of cycles if the sampling event is `CPU_CLK_THREAD_UNHALTED:THREAD_P`, or in terms of instructions if we use retired instructions `INST_RETIRED:ANY_P`. The abovementioned analysis is performed in more details in Section 3.5.

Alignment of profile groups. Sometimes, in order to create an execution region, we need more measurements that fit in a single profile group. Therefore, we need to create two or more profile groups G_1, G_2, \dots which hold all the required data. For example, the Top-down Micro-architecture Analysis (TMA), used with performance profiles, would require 17 groups, assuming 4 hardware performance counters per core [33]. However, those groups are unaligned because groups are always measured separately. The solution is to align performance profiles coming from those profile groups. We generalise this idea with an *alignment function* $\lambda(G_1, G_2, \dots)$ that creates a new group \mathcal{G} according to a recipe which follows some form of e.g. stretching, contracting, reducing, or averaging of performance profiles.

The alignment of profile groups is a non-trivial task for several reasons: (1) the sampling time is unevenly-spaced due to measurement errors of the sampling mechanism; (2) the values of a performance profile are inaccurate due to the imprecision and systematic errors of the Performance Monitoring Unit (PMU) and the non-deterministic events happening on a machine; (3) lengths of performance profiles differ, especially with cycles as the sampling event, because again of the non-deterministic events and measurement errors. The catch-all term “non-deterministic events” includes unexpected hardware interruption, dynamic frequency scaling of a processor, changes to processor affinity, branch prediction, memory prefetching, or any process rescheduling

and context switching by the Operating System, among others.

Nevertheless, in our work, we mainly utilise only single profile group and if otherwise, the groups are measured using much more stable retired instruction, than processor cycles.

3.5 Case study: cost of array slicing

In this section, we present one application of performance event profiles (PEP) to the cost analysis of data copy performed during array slicing. In MATLAB, each array slice requires a data copy as depicted in Figure 3.2 with the vectorised version of the code (`vec`). However, as with any costly operation, several questions arise such as: (1) how many cycles exactly takes data copy? (2) does the cycle cost change with the volume of copied data? An answer to question (1) gives information useful for taking a decision whether or not to vectorise a loop. If the vectorised loop requires a lot of explicit array slicing, then the benefit from using vector operations might be overshadowed by the cost of making data copies. Moreover, the question (2) asks if there is a fundamental difference in how MATLAB performs the data copy according to an increasing size of data. Differences in cost according to the size of data could indicate the use of various copying mechanisms by MATLAB (e.g. software prefetching, use of packed vector instructions) or that the machine is performing the copy differently (e.g by using a hardware prefetching).

We start by selecting benchmark codes for the analysis and performance events for building performance event profiles. Next, we repeatedly execute each benchmark with variable size of data and build performance event profiles for each execution. From the profiles, we measure the length of execution regions which perform data copy. Finally, we collect the information about duration of data copy and compute the cost of per-element copy.

Selecting benchmarks. Our cost analysis prefers simple codes with arithmetic operations and array slices, because in such codes, we can easily spot and measure the data copy regions. Therefore, we have selected the famous *STREAM* benchmark [126] and its extension *Bandwidth Benchmark* [127]. The goal of both benchmarks is to measure the maximal attainable bandwidth of memory load and store operations, which is perfectly in line with our goal of measuring the cost of data copy. Table 3.2 depicts 6 selected codes from both benchmarks which perform computation over vectors of size N . The last column in the table indicates the number of right-hand array slices in each code.

Selecting profile group. Data copies often occur before computation regions. Moreover, it can be also defined as a region with similar amount

Table 3.2: Selected programs from *Bandwidth Benchmark* [127] for the use in cost analysis of implicit data copy in array slicing.

Program	Code	#Slices
update	<code>a(1:N) = scalar .* a(1:N)</code>	1
add	<code>a(1:N) = b(1:N) + c(1:N)</code>	2
triad	<code>a(1:N) = b(1:N) + scalar .* c(1:N)</code>	2
daxpy	<code>a(1:N) = a(1:N) + scalar .* b(1:N)</code>	2
sdaxpy	<code>a(1:N) = a(1:N) + b(1:N) .* c(1:N)</code>	3
striad	<code>a(1:N) = b(1:N) + c(1:N) .* d(1:N)</code>	3

of load and store operations as Equation (3.2) indicate (data copy is about loading an element and storing it somewhere else). Therefore, for our task, we have defined a group G_{as} of profiles depicted in Equation (3.3). The group contains three profiles of: 128 B packed floating-point operations, load, and store operations. Moreover, because our analysis is concerned with processor cycles, all profiles from the group G_{as} use processor cycles as the sampling event. Thus, allowing us to measure the real cost of data copy.

$$\mathcal{E}(G_{as}) = (\text{CPU_CLK_THREAD_UNHALTED:THREAD_P}, \\ \{\text{FP_ARITH_INST_RETIRED:128B_PACKED_DOUBLE}, \\ \text{MEM_INST_RETIRED:ALL_LOADS}, \\ \text{MEM_INST_RETIRED:ALL_STORES}\}) \quad (3.3)$$

Measuring benchmarks. For the tests, we have selected M1 machine (consult Table A.1 for full specification) with MATLAB R2015b/R2018b running on a single thread because mPAPI is able to build performance event profiles only for the single thread execution. Furthermore, we test each benchmark with double precision floating-point data of size from 10 000 to 10 000 000 which represent from 0.23 MB to 228.88 MB of copied data in the case of `striad` benchmark. This big range of transferred data might help answer question (2) about existence of different copying mechanisms.

In order to get relevant measurements, each benchmark execution is repeated 30 times. The repeated execution is especially important with performance event profiles built using processor cycles, because of the existence of many sources of noise and measurement errors. Moreover, for the sampling threshold of the profile group G_{as} , we have selected a value of 100 000 processor cycles which gives an excellent trade-off between the size of the result profile file (only 128.4 MB), low impact on the values of measured performance events (see Figure 3.5), and the high level of details found in the profiles (see Figure 3.4). Moreover, the selected sampling threshold is well suited for our benchmarks run with data of size from 10 000 to 10 000 000.

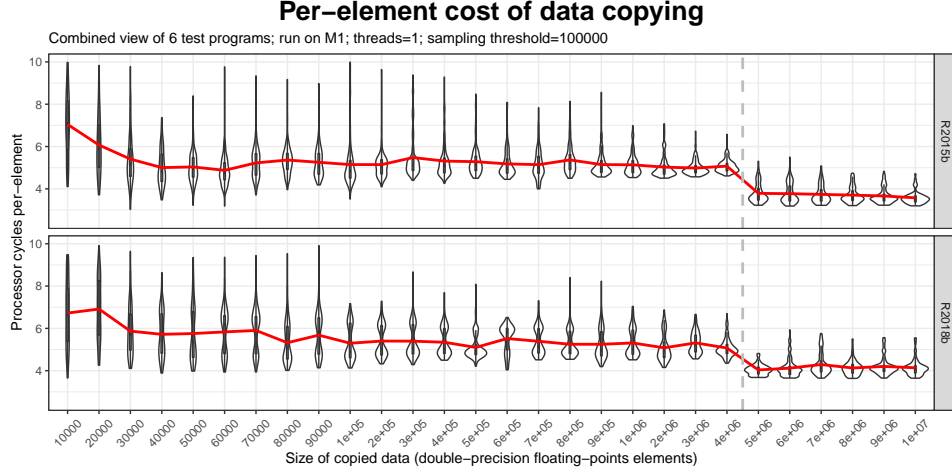


Figure 3.7: The per-element cost of performing data copy during *array slicing*. The results show the copy of huge volume of data (more than 4 000 000 elements) is performed more efficient in terms of processor cycles.

Finding data copy regions When comes to finding the data copy regions, we have two choices: (1) use the definition of a data copy region from Equation (3.2) and the profile group G_{as} containing measurements of load and store instructions; or (2) use a reference profile indicating when the data copy execution region starts and ends. Figure 3.2 shows that data copy regions occur before the computation region for benchmark *striad* which is the most complex one from our set of codes. Therefore, instead of using all three performance profiles from the group G_{as} , we can use only the profile of floating-point operations. From the profile, we extract the length of data copies by taking the period from the profile beginning, until the occurrence of the first floating-point operation. In this case, the profile acts as a reference profile (or a proxy profile) because by using it, we are actually obtaining information about the data copy which is more closely related to load and store operations instead of arithmetic operations.

Results. Collected lengths of data copies, from all benchmarks and for the same data size, were grouped and divided by the number of copied elements N and the number of right-hand array slices occurring in the tested benchmark code (from 1 to 3). Thus, the result indicates a per-element cost of performing array slicing with implicit data copy. Figure 3.7 presents a combined cost of all 6 codes from the study for two MATLAB versions and different data size. Results for each size of copied data are represented as violin plots [128] which shows the exact distribution of measurements. In Figure D.2 from Appendix D, we have placed detailed results where the per-element cost of data copy is depicted for each benchmark code separately.

The results unanimously show one recurrent pattern, the existence of two cost levels for data copy before and after copying up to 4 000 000 elements (30.52 MB). With copying more than 4 000 000 elements, the cost of per-element copy drops from 4 to 5 processor cycles. This indicates that MATLAB, both R2015b and R2018b, have two different methods for performing array slicing.

3.6 Conclusion

MATLAB, as a closed source environment, keeps information about how it analyses and compiles programs. Therefore, in order to understand how MATLAB executes programs, we have referred to hardware performance counters which describe the program execution on processors, bypassing the inner-workings of MATLAB altogether (see Figure 3.1). However, the sole values of performance events describing the program are not enough, because they do not consider the fact that a single expression in runtime environments, might in reality be a composition of many distinctive regions (see Figure 3.2). Therefore, the performance events must be collected over time to capture different parts of the program execution.

In this chapter, we have presented **Performance Event Profiles (PEP)** which is an innovative use of performance counters gathered in the sampling mode. Previously, researchers have used such profiles too, but their purpose were different ranging from analysis of server workload, through feedback-directed optimisations (FDO) to finding security vulnerabilities in programs. In this work, however, we correlate many profiles under a single *profile group* to assess what hides under the execution of expressions and statements in MATLAB code. We have achieved this by carefully selecting performance events showing intentions of particular parts of execution (called *execution regions*). The intent behind an execution region (e.g. computation or data copy) greatly differs from possible end results of the execution (e.g. cache misses, pipeline stalls).

In Section 3.3, we have introduced formally the performance profile, profile group, and their event domains. Furthermore, in the following subsections, we have shown how to measure them directly in MATLAB using our tool **mPAPI** (see Appendix B for more details). Moreover, we have described the benefits and problems coming from the selection of various sampling events and thresholds. Section 3.4 outlines the concept of *execution region* allowing to discover various parts of the program execution. Finally, Section 3.5 shows how using performance profiles measured over processor cycles can evaluate to the cost analysis of data copy.

Although, this chapter focuses solely on MATLAB, performance event profiles are applicable to other languages, runtime systems, and interpreters because the profiles measure execution from the perspective of a processor.

Chapter 4

Execution model for MATLAB

Résumé

Les profils d'événements de performance permettent non seulement d'analyser la performance des codes MATLAB mais, plus important encore, ils aident à découvrir les règles régissant l'exécution des programmes compilés par JIT. Dans ce chapitre, nous présentons une description formelle de la façon dont MATLAB exécute des expressions composées d'opérations sur des éléments de tableaux.

Nous commençons l'analyse en trouvant toutes les régions d'exécution à l'intérieur d'une seule expression. Chaque région représente un bloc d'instructions rempli d'opérations sur des éléments de tableaux programmées par le compilateur JIT de MATLAB et dont l'exécution est combinée. Le résultat d'une expression compilée par le JIT est un code machine généré dynamiquement qui est récupéré en mémoire, par l'intermédiaire d'un cache d'instructions. Par conséquent, en observant l'activité du cache d'instructions, nous pouvons facilement reconnaître combien de blocs d'instructions une seule expression génère et quand chaque bloc commence. Enfin, notre étude a révélé deux groupes de fonctions intégrées à MATLAB : 1) les fonctions uniques, par exemple `acos()`, `sum()` qui nécessitent un bloc entier d'instructions pour elles-mêmes ; et 2) les fonctions combinées, par exemple `cos()`, `times()` qui se combinent avec d'autres fonctions combinables pour former un bloc d'instructions partagé.

A partir de la décomposition connue des expressions en blocs d'instructions, nous avons proposé un modèle arborescent qui généralise les motifs observés. Le modèle proposé prédit, à partir d'une expression en entrée, le type et l'ordre des blocs d'instructions survenant pendant l'exécution du code.

Introduction

In MATLAB, programmers express element-wise computation using array operations or loops. Both code versions produce the same result, but they have different performance. For example, Figure 3.2 shows that loops in MATLAB use only scalar floating-point instructions on a single core, where array operations use vector instructions (e.g. with Intel SSE, AVX extensions) with possible multi-threading. Moreover, MATLAB with Just-In-Time (JIT)

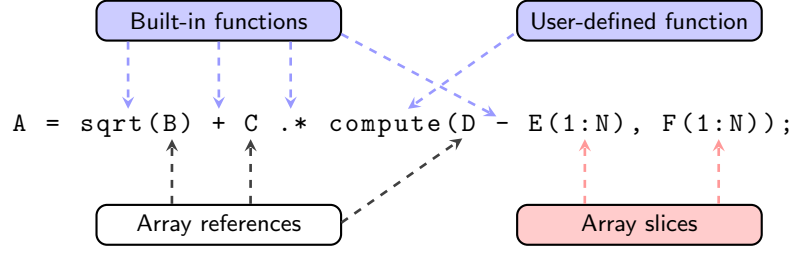


Figure 4.1: Components of MATLAB expressions with array operations.

compilation often compiles programs written with loops and array operations directly to the machine code. The advantages of array operations seem to overpower performance of loops, however, the diversity of execution modes leads to complex cases where the choice of the faster code version is non-trivial [1, 31].

The choice of how to write optimal code is even non-trivial for array operations alone. Without the JIT compiler, MATLAB would act as a plain interpreter which executes composition of array operations in a sequence. However, with the JIT compiler, a subset of operations from the composition are scheduled and executed together. The knowledge of which operations are JIT-compiled and how MATLAB schedules and executes array instructions is crucial for writing code with optimal performance.

In this section, we uncover and encode rules governing the execution of array operations with the dynamic compilation. These rules create an execution model for MATLAB expressions with a consideration of JIT compilation. The model includes information about the type and order of computation, function calls, and indexing of arrays mentioned in Section 4.1. Moreover, the model is based on the concept of an instruction block from Section 4.2 which is a set of instructions executed together. In Section 4.4, with the notion of instruction blocks, we are able to find out which built-in and user-defined functions can merge together and be inside the same instruction block. Then, we express how instruction blocks interact with each other in a form of instruction trees in Section 4.5. Finally, instruction trees allow us to make predictions about MATLAB codes.

This chapter is based on performance event profiles (PEP) and execution regions, introduced in Chapter 3, which are essential for building the model.

4.1 Scope of the execution model

We start our investigation by setting the scope of our execution model. MATLAB has an expressive language with a vast number built-in functions thus, expressing every possible detail in the model might be infeasible. Therefore, we focus on vectorised expressions consisting of four components: (1) built-in

functions; (2) user-defined functions; (3) array references; and (4) array slices, as depicted in Figure 4.1. These four components cover the majority of expressions in common MATLAB programs. However, we only consider expressions as part of assignment statements.

In an ideal world, every expression is JIT-compiled as one code region. However, in MATLAB, some of the abovementioned components divide the compilation of a single expression into multiple parts. For example, an array slice requires data copy which creates a new execution regions. In a similar fashion, built-in functions e.g. `sum`, `diff` split the compilation into three parts which evaluate: (1) function arguments, (2) the function itself; and (3) the rest of the expression. Moreover, every user-defined function behaves exactly like the non-JIT-compiled built-in function.

Our model describes built-in and user-defined functions containing array operations which perform element-wise computation. Other types of functions are matrix operations implementing linear algebra routines such as matrix multiplication (`*`, `mtimes()`) or solving systems of linear equations $Ax = B$ for x (`\`, `mldivide()`). Nevertheless, the presented execution model is capable of expressing matrix operations too. Moreover, we consider only single-thread execution in this part of the chapter, because predicting the multi-threaded execution requires additional information in the model which would unnecessarily complicate its description. Finally, we analyse two MATLAB versions, R2015b and R2018b, with a redesigned execution engine LXE which combines the interpreter and the JIT compiler into a monolithic structure [10].

4.2 Instruction blocks in JIT compilation

Without a Just-In-Time (JIT) compiler, MATLAB would be an interpreter which executes (interprets) instructions step by step. The interpreter fetches, decodes, and executes each instruction in isolation without any knowledge about future instructions. However, with the JIT compiler, MATLAB can defer evaluation of the instruction as long as possible (in the APL interpreter, this concept is known as *drag-along* [8, 129]). The delay often creates new optimisation opportunities for the JIT compiler, because the compiler carries information about future instructions. This leads to better instruction scheduling, register allocation and code optimisations.

However, the nature of instructions in MATLAB is questionable because without a formal specification, we are not sure about which parts of the language syntax are really executable. Instead, we make a simplification and assume that each call to a built-in or user-defined function is an instruction. We leave out array slicing as a specialised type of an instruction and further, we always consider the slicing separately.

Instruction block. We define an *instruction block* Γ as a program segment containing a set of instructions $\Gamma.instructions = \{\gamma_1, \gamma_2, \dots\}$ which MATLAB executes together. The instructions $\gamma_1, \gamma_2, \dots$ are calls to MATLAB built-in or user-defined functions. Instructions inside the block are stored as a set, because we are not concerned with their order of execution. Moreover, in many cases the real execution order would be hard to deduce and dependent on the compiler, compilation heuristics, and the target machine. Furthermore, the set of instructions is a multi-set capable of holding a few references to the same function.

If an instruction block Γ has multiple instructions $|\Gamma.instructions| > 1$, then we say the JIT compiler schedules and executes together instructions inside the block. Usually, the result of compilation of an instruction block is a regular execution region without time-varying behaviour, similarly to how MATLAB executes loops. Therefore, for these regions, we can use traditional metrics and models, e.g. the Roofline Model [32], Top-Down Micro-architectural Analysis Method (TMAM) [33], which do not consider time-varying execution by default.

Block attributes. It is convenient to extend the instruction block with additional information in a form of block attributes. Usually, instructions in the block ($\Gamma.instructions$) perform computation on data accessed through array references ($\Gamma.references$) or array slices ($\Gamma.slices$). The distinction between attributes for references and slices is important, because instructions directly use references to arrays, where array slices require data copy operations beforehand. Three standard block attributes include:

- $\Gamma.instructions = \{\gamma_1, \gamma_2, \dots\}$ a multiset (mset) of instructions in the block. Single γ_i represents a call to a built-in or user-defined function. We track all calls to the same function, because all of them performs computations (MATLAB has no seamless memoisation for functions).
- $\Gamma.references = \{a, b, c, \dots\}$ a set of array references to variables used in the computation. We track repeated uses of the same reference only once, because we are not concern about the data reuse at this stage.
- $\Gamma.slices = \{k, l, m, \dots\}$ a set of array slices used in the computation. Each array slice is just an object holding the information about a sliced variable (a, b, c) and the region of copied data.

Merging instruction blocks So far, we have mentioned how the JIT compiler executes instructions together forming an instruction block. However, it is not hard to imagine a code optimisation in which the JIT compiler executes together two instruction blocks. The new result block shares instructions, array references, and array slices coming from both input blocks. Algorithm 1

presents how the merge operation might take effect for instruction blocks with default attributes.

Algorithm 1 Function merging together two instruction blocks Γ_1 and Γ_2 .

```

function MERGEINSTRUCTIONBLOCKS( $\Gamma_1, \Gamma_2$ )
   $\Gamma_{result} \leftarrow \text{NEWINSTRUCTIONBLOCK}()$ 
   $\Gamma_{result}.instructions = \Gamma_1.instructions \uplus \Gamma_2.instructions$ 
   $\Gamma_{result}.references = \Gamma_1.references \cup \Gamma_2.references$ 
   $\Gamma_{result}.slices = \Gamma_1.slices \cup \Gamma_2.slices$ 
  return  $\Gamma_{result}$ 

```

In the case of $\Gamma.instructions$ attribute, the operation \uplus is a multiset sum which not only performs an union between two multisets, but it also adds repeated elements multiple times inside the multiset. The other two operations \cup are classical unions between sets.

4.3 Detecting instruction blocks

A successful execution of an instruction block indicates that the JIT compiler fetches, decodes, schedules, and executes every instruction from the block. The first stage, fetch and decode, requires retrieving instructions for the execution from the instruction cache. This process usually generates several performance events on the processor, e.g. `L2_RQSTS:ALL_CODE_RD` if the instructions are in the L2 instruction cache, or `OFFCORE_RESPONSE_*:DMND_CODE_RD` if they are requested from the external memory. Therefore, by observing changes in performance events related to instruction fetch and decode, we can find the beginning of the instruction block. This is also true for instruction blocks with only one instruction $|\Gamma.instructions| = 1$. In that case, MATLAB behaves like a pure interpreter executing instruction by instruction. Moreover, JIT compilers have finite granularity of compilation which means the compilers work on portions of the code instead of whole programs (e.g. method or trace-based compilation focusing on loops) [130, 131]. Hence, programs of reasonable size have more than one instruction block.

With the use of *performance event profiles* (PEP) introduced in Chapter 3 and a careful selection of a single performance event which indicates fetch and decode stage, we can find the start point of an instruction block. For the performance event candidates, we have consider events measuring activities related to instruction fetching such as cache misses or hits. Some event candidates collected in Table 4.1 are interconnected with each other. From their descriptions, we can easily spot direct and indirect relations between events. For example, cache misses propagate throughout many levels of cache memories and instructions entirely missing from the cache can cause misses in the instruction translation-lookahead buffers (iTLB).

Table 4.1 presents performance events which consist of random information

Table 4.1: Performance events related to instruction fetch and decode on *Skylake* and *Coffee Lake* microarchitectures [132]. Events description comes from the Intel® Processor Event Reference [133]. Not every performance event candidate can indicate the start of instruction blocks.

Performance event	Description	Metric
FRONTEND_RETIRED:DSB_MISS	Counts retired Instructions that experienced DSB (Decode stream buffer i.e. the decoded instruction-cache) miss.	✗
FRONTEND_RETIRED:ITLB_MISS	Counts retired Instructions that experienced iTLB (Instruction TLB) true miss.	✗
FRONTEND_RETIRED:L1I_MISS	Retired Instructions who experienced Instruction L1 Cache true miss.	✗
FRONTEND_RETIRED:L2_MISS	Retired Instructions who experienced Instruction L2 Cache true miss.	✓
ICACHE_16B:IFDATA_STALL	Cycles where a code line fetch is stalled due to an L1 instruction cache miss. The legacy decode pipeline works at a 16 Byte granularity.	✓
ICACHE_64B:IFTAG_HIT	Instruction fetch tag lookups that hit in the instruction cache (L1I). Counts at 64-byte cache-line granularity.	✗
ICACHE_64B:IFTAG_MISS	Instruction fetch tag lookups that miss in the instruction cache (L1I). Counts at 64-byte cache-line granularity.	✓
ICACHE_64B:IFTAG_STALL	Cycles where a code fetch is stalled due to L1 instruction cache tag miss.	✓
ITLB_MISSES: MISS_CAUSES_A_WALK	Counts page walks of any page size (4K/2M/4M/1G) caused by a code fetch. This implies it missed in the ITLB and further levels of TLB, but the walk need not have completed.	✓
ITLB_MISSES: WALK_COMPLETED	Completed page walks (2M and 4M page sizes) caused by a code fetch. This implies it missed in the ITLB and further levels of TLB. The page walk can end with or without a fault.	✓
L2_RQSTS:ALL_CODE_RD	Counts the total number of L2 code requests.	✓
L2_RQSTS:CODE_RD_MISS	Counts L2 cache misses when fetching instructions.	✓
OFFCORE_RESPONSE_*: DMND_CODE_RD	Counts both cacheable and non-cacheable code read requests.	✓

Listing (4.1) Interpreted computation expressed in *two statements* with an explicit intermediate variable `tmp`.

```
1 tmp = A1 + A2(1:LEN_1D);
2 R = tmp .* A3;
```

Listing (4.2) Dynamically compiled computation collapsed into just *one statement*.

```
1 R = (A1 + A2(1:LEN_1D)) .* A3;
```

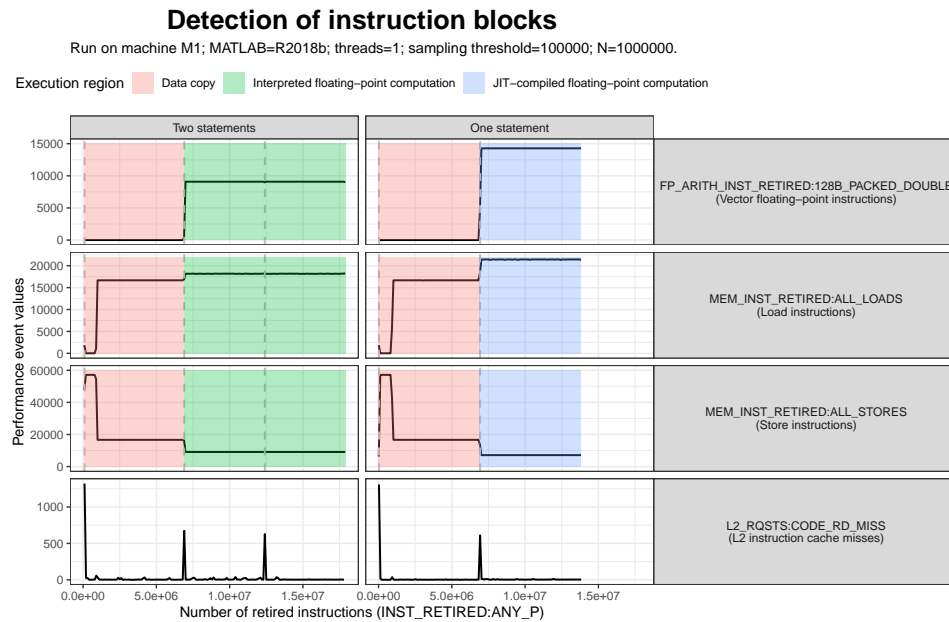


Figure 4.2: Detection of instruction blocks using instruction cache misses. Execution region of interpreted floating-point computation actually contains two instruction blocks (addition followed by multiplication).

(`FRONTEND_RETIRED:DSB_MISS` or `ICACHE_64B:IFTAG_HIT`), but also events containing measurements perfectly correlating with start and end points of execution regions (`L2_RQSTS:CODE_RD_MISS` or `OFFCORE_RESPONSE_*:DMND_CODE_RD`). Figure 4.2 depicts how spikes on the profile of `L2_RQSTS:CODE_RD_MISS` relate to execution regions. We assume, that each spike indicates the start of a new instruction block, because the spike represents the load of instructions designated for the execution. With the assumption, it is easy to see on the left graph in Figure 4.2 that relaying only on execution regions is not enough to distinguish between two separate, but identical computations. In the plot, execution region of *interpreted floating-point computations*, marked in green, actually consists of two separate instruction blocks. Moreover, with correctly detected instruction blocks, this figure also indicates MATLAB JIT compiler granularity of compilation which is a single statement.

4.4 JIT compilation of functions

The core of any instruction block is a set of instructions compiled together by the JIT compiler. However, not every MATLAB function forms a multi-instruction block. Complex functions or calls to external libraries generate separate instruction block(s) because the JIT compiler is not able to emit a machine code for them, and instead, emits only calls to those functions.

In this section, we use the detection scheme from Section 4.3 to analyse the number of instruction blocks inside compositions of MATLAB functions. For the task, we prepare and analyse specialised codes which combine several MATLAB functions into various compositions. Later, during the analysis, we observe the number of instruction blocks generated by those codes (just like in Figure 4.2). If a composition of functions generates only one block, it means the JIT compiler executed the composition by emitting single piece of machine code. We apply the same analysis to both, MATLAB built-in and user-defined functions.

The ability to predict how JIT compiler schedules and executes instructions, allow to reorganise an expression in a way that the new expression executes faster. This is possible, because new order of evaluation might reduce the number of instruction blocks generated from the expression. In other words, the expression with lower number of blocks indicates better code scheduling by the JIT compiler. One of the goals of our model is to predict the number and the content of instruction blocks in the program. With the complete picture on the computation, we can start working on specialised transformations which help the JIT compiler to better schedule the program.

4.4.1 Built-in functions

The analysis of built-in functions starts with running the three code patterns depicted in Table 4.2 for each function. Every pattern is a composition of

Table 4.2: Three detection patterns used for the detection of dynamic (JIT) compilation of MATLAB functions. The presented patterns use functions `cos` and `atan2` as an example.

Detection pattern	Unary	Binary
Self-composition	<code>cos(cos(A))</code>	<code>atan2(atan2(A1, A2), A3)</code>
Composition <code>plus:+</code>	<code>cos(A1)+cos(A2)</code>	<code>atan2(A1, A2)+atan2(A3, A4)</code>
Arguments <code>plus:+</code>	<code>cos(A1+A2)</code>	<code>atan2(A1+A2, A3+A4)</code>

at least two functions, because only then we are able to observe if the JIT compiler creates one instruction block for these functions. However, this does not mean a single instruction is not JIT-compiled, we just are not able to observe it with this approach.

The first pattern is a self-composition `f(f(A))` which is the simplest way for one function to create a complex expression. The next two patterns compose the function `f` with addition operator `+/plus` which is JIT-compiled (we have verified this by observing if a composition of many additions is translated into a single basic block of machine code with packed double-precision floating point instructions `addpd` on x86 architecture; in order to lookup and analyse the machine code, we have used Intel VTune [99]). The second pattern `f(A)+f(B)` tests if the `plus` composes well with functions `f` as arguments. Finally, the third pattern `f(A+B)` validates if the function `f` executes with a complex expression as an argument inside a single instruction block. Table 4.2 presents detection patterns for unary and binary functions.

The self-composition pattern from Table 4.2 is not applicable to every function. Consider reduction operations `sum` or `prod`. Their first execution in the self-composition reduces the input array by one dimension (e.g. a vector reduces to a scalar). Therefore, the second call to the function works on data with a significantly reduced size. Consequently, if the self-composition of `sum` or `prod` generates two instruction blocks, then the second block is very small, thus, hard to compare (it might get lost between measurement samples). Therefore, we do not use the self-composition for these two functions.

For the test of unary and binary built-in functions in MATLAB, we have selected elementary arithmetic operations, rounding functions, modulo division, trigonometric functions, exponents and logarithms, reductions, Fast-Fourier Transformations, and more. Table 4.3 presents which MATLAB functions can be combined with others (*combinable functions*) and which always generate an additional instruction block (*single functions*) just for themselves. Typical examples of *single functions* include `fft` and `mtimes` which are delegated to external libraries, FFTW [134,135] and Intel MKL [5] respectively. For the *combinable functions*, prime examples include basic arithmetic operators, e.g. `plus`, `times`, `ldivide`, which are easy to combine and

Table 4.3: List of built-in functions which require single instruction block (*single functions*) or merge with other instructions into a common instruction block (*combinable functions*). All functions were tested on machine M1 and MATLAB R2015b/R2018b.

Single functions	Combinable functions
acos, acosh, asin, atanh, :, colon, ', ctranspose, cumprod, cumsum, det, diff, eig, expm1*, fft, fliplr, gamma, ifft, log, log10, log1p, log2, max, mean, min, mtimes, nextpow2, norm, ones, .^, power, prod, rand, randn, sqrt, sum, zeros	abs, asinh, atan, atan2, ceil, cos, cosh, exp, expm1 [†] , fix, floor, imag, .\, ldivide, -, minus, mod, +, plus, pow2, ./, rdivide, real, rem, round, sign, sin, sinh, tan, tanh, .*, times, .', transpose, uminus, uplus

* Only R2015b.

[†] Only R2018b.

schedule together. The obtained results are consistent for MATLAB R2015b and R2018b, except for the function `expm1` which is dynamically compiled only on version R2018b. Furthermore, the presented testing procedure is general and easy to extend to more functions.

4.4.2 User-defined function

MATLAB programmers can create their own user-defined functions which raises two questions about how JIT compiler manages them: (1) is the body of an user-defined function JIT-compiled? (2) is an expression which uses user-defined functions JIT-compiled too? A positive answer to question (1) means the JIT compiler is method-based with a whole function as the compilation unit [131], or at least the compiler compiles statements in the function body, but separately. On the other hand, question (2) explores capabilities of the JIT compiler in merging instruction blocks created from user-defined functions with other blocks. In this section, we analyse both questions (1) and (2), about user-defined functions, using the same analysis and detection patterns as in Section 4.4.1.

JIT compilation of user-defined functions. In order to test if the JIT compiler also compiles the body of user-defined functions, we have prepared a set of test routines (similar to test functions in Figure 4.3), which consist of only combinable functions from Table 4.3. Therefore, we know that the bodies of these test functions can be JIT-compiled. Moreover, apart from testing single assignments in the test function body, we have prepared test functions containing multiple JIT-compiled statements. Therefore, successful

<pre> 1 function R=un_1_block(A) 2 R = A + A; 3 end </pre>	<pre> 1 function R=un_2_blocks(A) 2 R = log10(A) + A; 3 end </pre>
<pre> 1 function R=bin_1_block(A, B) 2 R = A + B; 3 end </pre>	<pre> 1 function R=bin_2_blocks(A, B) 2 R = log10(A) + B; 3 end </pre>

Figure 4.3: Examples of test functions which are JIT-compiled to one or two instruction blocks. With the use of *single function* `log10` (see Table 4.3) in `*_2_blocks()`, we force the JIT compiler to issue two instruction blocks.

compilation of such functions would indicate that the JIT compiler is able to combine and optimise expressions coming from multiple statements (like a method-based compiler would do). As an example, consider Listing 4.1 with a small program (which could be a function body) containing two statements which can be optimised and merged into one statement as in Listing 4.2.

Our results, obtained from running test functions, prove two things: (1) JIT compiler in MATLAB compiles the body of user-defined functions, however, (2) the compiler will not merge, schedule, nor execute together multiple statements from a user-defined function. Therefore, although the JIT compiler compiles the function body, the granularity of compilation still stays at a level of a single statement, hence, an user-defined function with N statements always compiles to at least N instruction blocks.

JIT compilation of expressions with user-defined functions. Considering that the MATLAB JIT compiler handles user-defined functions, we move to testing how dynamic compilation affects expressions with calls to those functions. For this analysis, we treat user-defined functions just like built-in functions in Section 4.4.1. Moreover, we reuse the same testing procedure as before and the same set of detection patterns from Table 4.2.

Figure 4.3 presents four simple user-defined functions which we use for testing. Functions `*_1_block` contain only `plus` which generates one instruction block. On the other hand, functions `*_2_blocks` combine `plus` with `log10` which requires separate instruction block, thus, the functions consist of two instruction blocks.




Our results can be summarise into two points: (1) the user-defined functions are not merged with `plus` function; and (2) the functions force a sequential evaluation of arguments, from left-to-right. In other words, user-defined functions are never merged with other instructions and they do not share a single instruction block. Moreover, each expression passed as an argument to the user-defined function, creates at least one instruction block. Therefore, single call to the user-defined function with N expressions

passed as arguments creates at least $N + 1$ instruction blocks with at least 1 instruction block for the body.

4.5 Instruction tree

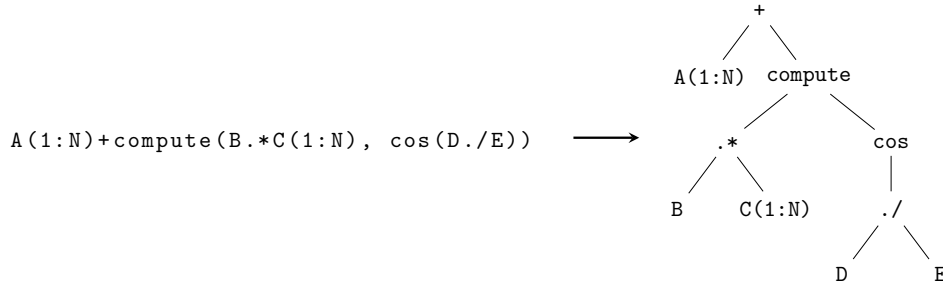
In MATLAB, expressions consist of many instructions grouped into one or more instruction blocks. The role of the Just-In-Time (JIT) compiler is to schedule block of instructions for the combined execution. From our perspective, the execution of a single instruction block is an atomic action, because, from execution regions, we are not able to distinguish particular instructions. Therefore, we consider each instruction block as an atomic entity too. However, we care about how those blocks were formed and what is the execution order of particular blocks in an expression. Thus, we propose a new tree-based model for MATLAB expressions called the *instruction tree* which tracks and predicts how instructions merge into instruction blocks and in what order these blocks execute.

The instruction tree represents the execution order of instruction blocks and the use of variables, similar to how the abstract-syntax tree (AST) represents instructions and variables. However, unlike the AST, inner nodes depict instruction blocks instead of single instructions. Moreover, the instruction tree indicates possible unions of instruction blocks.

Components of the instruction tree. The instruction tree consists of one type of inner nodes:  *instruction block* — an execution block of one or more instructions. Moreover, the tree uses two different kinds of nodes  *array reference* and  *array slice* indicating the use of variables. Even though, the array slice could be considered as an instruction because it performs a data copy. However, array slice never merges with an instruction block, thus, we consider the slice as a separate entity.

Actually, as we will see in the next section, an input expression can be represented by a set of instruction trees, each with a different amount of instruction blocks. This fact is especially visible at the first stage of building instruction tree, when we translate each node of the input abstract-syntax tree (AST) to a block with only one instruction. Later, the tree shrinks because we merge together nodes in instruction blocks, thus, reducing their number.

The initial instruction tree, with instruction blocks containing only one instruction each, is a *maximal instruction tree*, because the tree could have blocks which can be merged. On the other hand, a *minimal instruction tree* has minimal number of instructions blocks, where no new merges are possible. Our model is concern with only the minimal trees, because they are the true representation of the execution of JIT-compiled expressions in MATLAB. In the next section, we present an algorithm for obtaining minimal instruction

Figure 4.4: **Step 1:** Conversion of an expression to AST.

trees from MATLAB expressions.

4.5.1 Building minimal instruction tree

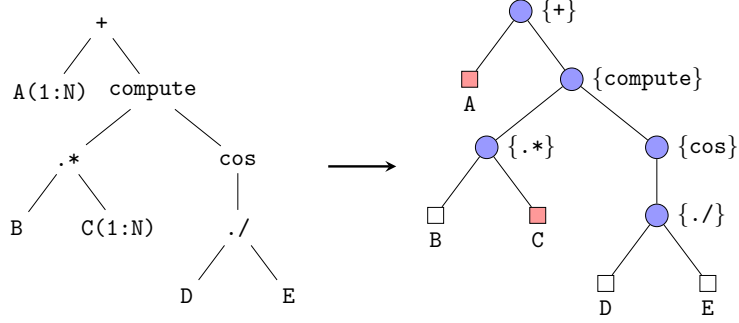
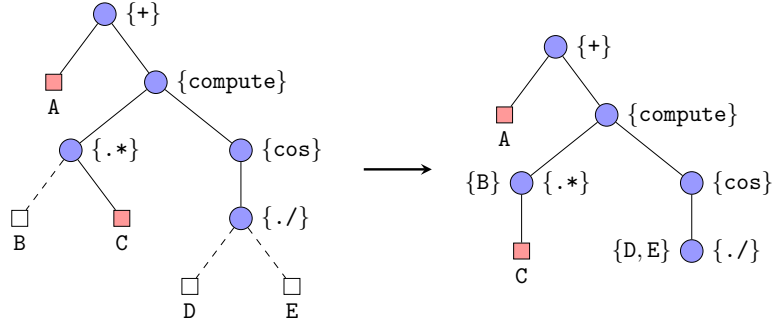
This section presents how to translate an input MATLAB expression to a minimal instruction tree, which indicates the type and execution order of instruction blocks. The translation consists of 4 steps.

Step 1: Expression to AST. Our method works on expressions represented with abstract-syntax trees (AST) which are a good starting point, because they indicate the instruction order and the use of variables, a two key components of our execution model. Figure 4.4 presents an example MATLAB expression with its AST representation. Although, the exact details and approaches to building an AST might differ, we assume that the AST consists of separate nodes for each function call and variable reference (or slice). Moreover, arguments of a function call are subtrees with an order corresponding to their position (from left to right).

In order to create AST from an expression, we use our source-to-source compiler **HU!M** described in details in Chapter 6. The compiler uses AST as the main intermediate representation (IR) for code analyses and transformations. Moreover, **HU!M** implements the execution model described in this chapter.

Step 2: AST to instruction tree. The next step is a conversion from the AST to an initial instruction tree depicted in Figure 4.5. The translation maps each AST node to a corresponding node in the universe of instruction trees. Therefore, every reference to a variable maps to □ *array reference*, indexing of a variable changes to □ *array slice*, and every operation or function call translates to an ● *instruction block*. In the Figure 4.5, on the right of an instruction block, the set indicates which instructions are inside the block (the $\Gamma.instructions$ attribute defined in Section 4.2).

The obtained tree could be named a *maximal instruction tree*, because at this point every node represents an instruction block with just one instruction

Figure 4.5: **Step 2:** Conversion from an AST to the instruction tree.Figure 4.6: **Step 3:** Removing leaves with array references, but keeping the information inside instruction nodes.

and none of the blocks being merged together. Only in the next few steps, the maximal instruction tree will gradually transform into the minimal instruction tree.

Step 3: Removing of array reference leaves. Before merging instruction blocks, we remove array reference leaves from the instruction tree for two reasons: (1) they do not generate any execution regions on their own; and (2) semantically, they belong to the instruction block where references are used directly in the generated machine code to perform computations. Therefore, we save the information about references inside instruction blocks in the $\Gamma.references$ attribute. Figure 4.6 depicts the removing of array reference nodes. Furthermore, we still display the information about array references in the instruction trees, but on the left of instruction block nodes.

Step 4: Merging instruction blocks. The last stage is a repetitive process which merges pairs of instruction blocks as long as there exists any pair of combinable blocks. Therefore, the result is a minimal instruction tree consisting of a minimal possible amount of instruction blocks. The result

tree indicates the order of execution regions occurring during the execution of MATLAB expressions.

Algorithm 2 Building minimal instruction tree by repetitive merging of instruction blocks inside an instruction tree.

Input: the root node of the initial instruction tree

Output: minimal instruction tree

```

1: function CANMERGE(node)
2:   correctInst  $\leftarrow$  node.instructions  $\subseteq$  combinableFunctions ▷ Table 4.3
3:   return ISINSTRUCTIONBLOCK(node)  $\wedge$  correctInst

4: function BUILDMINIMALTREE(node)
5:   revChildren  $\leftarrow$  REVERSELIST(node.children) ▷ Visit from right-to-left
6:   for child in revChildren do
7:     BUILDMINIMALTREE(child) ▷ Recursive visit of the tree
8:   if CANMERGE(node)  $\wedge$  CANMERGE(node.parent) then
9:     if  $\neg$ HASRIGHTSIBLING(node) then
10:      node.parent  $\leftarrow$  MERGEINSTRUCTIONBLOCKS(node, node.parent) ▷ Algo. 1
11:      ATTACH(node.children, node.parent) ▷ Attach children to the grandparent
12:      REMOVEFROMPARENT(node)

```

The main function BUILDMINIMALTREE from Algorithm 2 is a recursive method which traverses the instruction tree and finds candidate instruction blocks for merging. The routine traverses the tree in a post-order, but with children visited right-to-left (lines 5–7). The reason for the reversed visit of children is the evaluation order of arguments in MATLAB which is left-to-right. Hence, MATLAB evaluates expression $e1 + e2$ starting with arguments $e1$, $e2$ and finishing with the $+$ operator which creates an evaluation sequence: $e1$, $e2$, $+$. With the standard post-order traversal and left-to-right visiting of children, we would never merge $e1$ with $+$ (even if possible), because the second expression $e2$ stands on the way — $e2$ evaluates in between $e1$ and $+$. However, if we visit children right-to-left, then we could merge $e2$ with $+$ and give to $e1$ an opportunity to merge with a newly created block of instructions $\{e2, +\}$. Therefore, the merge of a node with its parent is possible only when the node has none of right siblings (line 9).

The presented perspective on merging nodes (instruction blocks) is related to the structure of the instruction tree which encodes the evaluation order of MATLAB operations. However, the lack of right siblings of a node is not the only condition required for merging instruction blocks. The other condition, even more important, is if both instruction blocks contain only *combinable instructions* which can merged with each other (please refer to Table 4.3 in the Section 4.4.1). The function CANMERGE from Algorithm 2 encapsulates the condition and it is used in line 8. Moreover, the CANMERGE can work only on instruction blocks, hence, the use of the ISINSTRUCTIONBLOCK predicate. Finally, the procedure MERGEINSTRUCTIONBLOCKS from Algorithm 1 merges

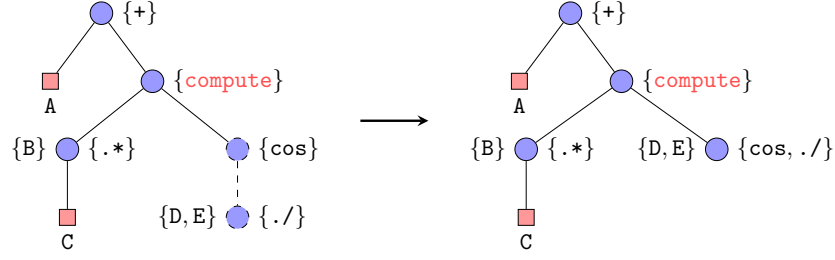


Figure 4.7: **Step 4:** Merging of instruction blocks. User-defined function `compute` prevents further merging of blocks. The right hand side tree is the minimal instruction tree.

two valid blocks and the subsequent operations rebuild the structure of the instruction tree by connecting child nodes (line 11) and removing one of the merged nodes (line 12).

Figure 4.7 depicts merging of two instructions blocks `{cos}` and `{./}` which contain only *combinable functions*. The procedure starts from the right-most child, the `{./}`, accordingly to the post-order with right-to-left visiting order. The merging of the blocks is valid because `{./}` has no siblings. However, none of the future unions are possible because of the user-defined function `compute`. We mark `compute` in red to highlight the fact that the function is not *combinable*, thus, preventing any future block merging.

Table 4.4 presents 8 more examples of expressions with their corresponding minimal instruction trees. Each minimal tree was obtained by applying Algorithm 2. Functions highlighted in red indicate non-*combinable* functions which prevent further merging of instruction blocks. This behaviour can be seen in the case of expressions 1, 2 and 3, where *single functions* `sum`, `sqrt`, and `.^` repetitively, prevent merging with other instruction blocks. However, it is not the case for expression 6, where the `log` function is not the barrier, but the array slicing of `C`. Furthermore, Expression 3 is an interesting example, because it collapses to a single node which means, the expression is perfectly scheduled by the JIT compiler. In other cases 5, 7, and 8, the existence of array slices as node's right sibling is the limiting factor of blocks merging.

4.5.2 Predicting execution from minimal instruction tree

Minimal instruction tree is a compact representation of execution regions occurring while executing an input MATLAB expression. In the tree, ● *instruction blocks* indicate the type of executed operations while ■ *array slices* mark data copies of arrays. Moreover, following a natural order of expression evaluation, the post-order, we obtain an arrangement of execution regions called the *instruction chain*, depicted in Figure 4.8. The result chain is just a flatten instruction tree.

Table 4.4: Examples of expressions with their minimal instruction trees. Instructions highlighted in red indicates functions which prevent instruction blocks from merging.

№	Expression	Minimal instruction tree
1	<code>sum(round(A))</code>	
2	<code>floor(A) + sqrt(fix(B .* C))</code>	
3	<code>floor(A) + sin(fix(B .* C))</code>	
4	<code>exp((A.^D+B.^E) ./ (C.^F))</code>	
5	<code>A(1:N) .* atan2(B(1:N), C)</code>	
6	<code>log(A) + B + C(1:N)</code>	
7	<code>fix(A(1:N))+(B(1:N).*C(1:N))</code>	
8	<code>A(1:N) + (B(1:N) + C(1:N))</code>	

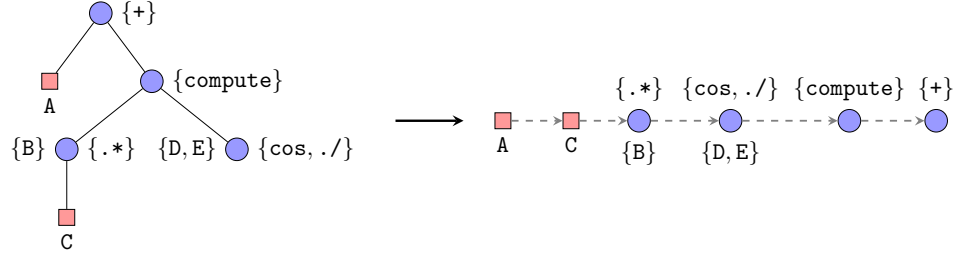


Figure 4.8: The result of traversing the instruction tree in a post-order is the instruction chain — a direct prediction of execution regions and their order.

Figure 4.9 presents how the instruction chain predicts and matches with a real measurement of the example expression. The performance profiles from the figure illustrate 6 distinctive execution regions: *data copy* $\times 2$, *vector floating-point computation* $\times 3$ (one for the `compute` function which performs vector addition) and one long *scalar floating-point computation*. Each node in the instruction chain matches with exactly one execution region. However, as seen in Figure 4.9, the instruction chain does not indicate the length of particular execution regions. In order to encode the length of instruction blocks and their execution regions, we would need to extend the instruction tree model with additional information about the execution of particular instructions and their composition inside instruction blocks. However, we plan to do so in the future.

4.6 Conclusion

This chapter introduces a methodology for the detection and modelling of the execution of the expressions in MATLAB. We have presented a tree-based execution model which not only encodes execution order of instructions and array slices inside expressions, but it also indicates when the MATLAB Just-In-Time (JIT) compiler schedules those instructions for combined execution as part of *instruction blocks*. Our methodology is entirely based on performance event profiles (PEP) introduced in Chapter 3. With these profiles, we are able to determine which instructions are in an *instruction block* and match them with a corresponding *execution region*.

To sum up, in this chapter, we have made the following contributions:

- Introduction of *instruction blocks* which consist of one or more MATLAB combinable functions (presented in Section 4.2). In Section 4.3, we describe a methodology for recognising instruction blocks inside performance event profiles and matching them with particular execution regions. To achieve this task, we have used a performance event `L2_RQSTS:CODE_RD_MISS` which indicates when MATLAB fetches instruc-

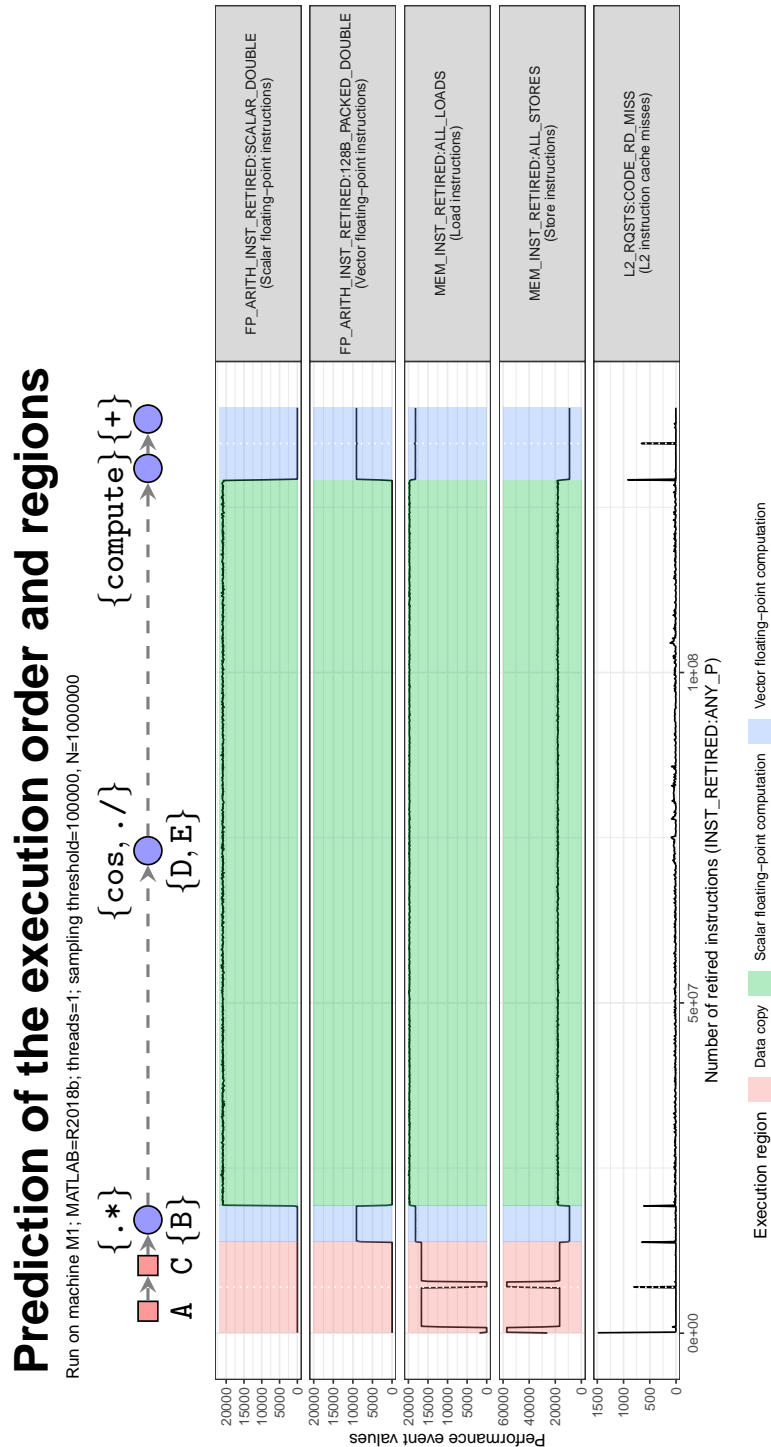


Figure 4.9: The flatten instruction tree in a form of the chain predicts the order and the type of instruction blocks and execution regions for the expression: $A(1:N) + \text{compute}(B.*C(1:N), \cos(D./E))$.

tions for the execution, which points to the start of a new instruction block (see Table 4.1 for other performance event candidates used for solving this problem).

- Analysis of which MATLAB built-in and user-defined functions can be combined with others to form *instruction blocks* (described in Section 4.4). The analysis works by testing how many instruction blocks the JIT compiler generates for various compositions of built-in and user-defined functions. Therefore, we divide functions into two groups of *single*, and *combinable functions* which are able to be merged with others (Table 4.3 presents built-in functions and the category they belong to).
- Introduction of a tree-based execution model consisting of *instruction blocks* and *array slices* as the tree nodes (presented in Section 4.5). The model, named *instruction tree*, represents the execution order of instruction blocks and also helps to visualise the potential candidate blocks for merging. If the instruction tree lacks of any more merging candidates, we say the tree is *minimal*. The *minimal instruction tree* displays the final representation of how the MATLAB JIT compiler will group, schedule, and prepare instructions (Figure 4.9 depicts example prediction). Finally, Algorithm 2 and our methodology described in section Section 4.5.1 show, step by step, how to obtain the *minimal instruction trees* for MATLAB expressions.

The presented results form a basis for code transformations which reorder, combine or split expressions in order to change (usually minimise) the number of instruction blocks issued by the JIT compiler. Knowing the rules of how the JIT compiler works, and changing the expressions accordingly, we could improve the performance of the expression execution.

Although, in this chapter, we focus entirely on analysing MATLAB expressions, our future work includes application of our methodology and our model to other Just-In-Time environments, such as Julia [136] and PyPy JIT compiler [137, 138] for Python. Moreover, we plan to extend our model with information on particular instruction blocks such as their performance (e.g. attainable memory bandwidth, IPC), and some information about how the JIT compiler schedules instructions on many threads.

Chapter 5

Code transformations for array operations

Résumé

Avec un modèle d'exécution des expressions, nous avons une image complète de la façon dont MATLAB effectue les calculs. Sachant cela, nous pouvons maintenant concevoir des transformations de code qui exploitent les points faibles et les points forts du compilateur JIT dans MATLAB. Dans ce chapitre, nous décrivons plusieurs transformations de code, qui toutes augmentent les performances des programmes MATLAB.

La première transformation, le repacking des sections de tableaux, utilise le fait que la copie de données (section de tableaux) crée un bloc d'instructions. Souvent, la copie est effectuée entre deux opérations de tableaux. Ainsi, elle divise ces opérations en deux blocs d'instructions distincts alors qu'on pourrait autrement n'en conserver qu'un seul. Le rôle du reconditionnement des sections de tableaux est de détecter ces copies et de les exécuter avant les calculs proprement dits.

La simplification des plages d'indexation (fonctions d'accès aux éléments de tableaux) est la deuxième transformation proposée. Elle tire parti du fait que le compilateur JIT de MATLAB génère un code très efficace pour le découpage en sections de tableaux si ses paramètres sont des nombres entiers. La transformation remplace les expressions d'indexation complexes par des plages de nombres beaucoup plus simples.

Enfin, l'optimisation de la boucle guidée par le profil permet de tester empiriquement si la vectorisation de la boucle est bénéfique. La procédure de profilage teste la boucle et sa forme vectorisée à partir de seulement quelques exécutions, mais chacune avec une taille croissante des données d'entrée. Ensuite, les résultats sont extrapolés pour des données d'entrée beaucoup plus importantes. Cela permet de réduire le temps nécessaire au profilage.

Introduction

In Chapter 4, we have created an execution model for evaluating expressions with array operations in MATLAB. The model not only gave us an

understanding of how MATLAB executes expressions in terms of *instruction trees*, but it also gave us a framework for thinking about possible code transformations and improvements on MATLAB.

In this section, we present a set of code transformations and improvements to the performance of MATLAB programs with array operations: (1) reimplementing the array slicing by introducing a dynamic array slicing and by removing redundant memory initialisations (Section 5.1); (2) repacking of array slices into new variables allowing the Just-In-Time (JIT) compiler to merge operations (Section 5.2); (3) simplifying and transforming numeric ranges inside array slices (Section 5.3); and (4) investigating the idea of Profile-Guided Optimisation (PGO) for loop vectorisation in MATLAB.

5.1 Redesigning array slicing

Computations often require only a subset of an array, e.g. odd elements or non-zero elements. To extract the subset of an array, MATLAB has three indexing methods: (1) positional indexing with exact coordinates of elements; (2) linear indexing which treats a multi-dimensional array as one-dimensional; and (3) logical indexing with a boolean condition. Positional indexing and logical indexing are also known as *array slicing*, a common operation in array languages including APL, Julia, Octave and many others. In MATLAB, function `subsref()` implements array slicing for vectors, cell arrays and object indexing for classes. However, the array slicing is not a fast operation because it always makes a copy of extracted elements.

5.1.1 Dynamic array slicing

In MATLAB, expression `A(1:end)` is equivalent to writing just `A` (`end` indicates the number of elements in a given dimension, here `length(A)`). However, the array slice here is not only redundant, but it generates unnecessary operations, e.g. data copy, out-of-bound checks, type checks. While MATLAB programmers do not write code like that, they do write array slices `A(1:n)`, `A(m:end)`, and `A(m:n)`. Furthermore, MATLAB compiler `Mc2Mc` often creates similar slices during loop vectorisation [1].

Slices `A(1:n)`, `A(m:end)`, `A(m:n)` are valid, however, they become redundant when `n==length(A)` and `m==1`. In a dynamic language like MATLAB, it is not possible to test if `n==length(A)` and `m==1` are true in every case without running the program. Therefore, we propose to delay the test until the program execution and select the array slice or reference, accordingly.

Listing 5.1: Dynamic array slicing with `d_slice()` which selects either array reference or array slice depending on the indexing range `[from:step:to]`.

```

1 function result = d_slice(array, from, step, to)
2   if from == 1 && step == 1 && numel(array) == to
```

```

3   result = array;                % Select array reference
4   else
5       result = array(from:step:to); % Select array slice
6   end
7   end

```

For the dynamic selection of array slices or references, we have prepared a function `d_slice()` depicted on Listing 5.1. The function tests if the range `[from:step:to]` extracts the whole array or just a slice and it returns either the array reference or slice. With the use of our **HU!M** compiler, described in Chapter 6, we can automatise the replacement of every array slice by the corresponding call to the `d_slice()`. For example, the array slice `A(m:end)` changes into `d_slice(A, m, 1, numel(A))`. Moreover, it is possible to reimplement `d_slice()` for multi-dimensional arrays.

Call to `d_slice()` does not incur an overhead because the function only once checks if the indexing expression covers the whole array. However, `d_slice()` is an user-defined function, therefore, it does not improve the JIT compiler capabilities of merging instructions together into a single instruction block. Nevertheless, the function eliminates unnecessary array slicing and implicit data copies during the runtime which is still highly desirable.

Skipping static code analysis. The approach of deferring certain tests to the runtime (or whole code transformations), bypassing the static code analysis completely, applies to other operations like array transposition (`(c)transpose`) or data replication (`repmat`). Both operations depend on the size of their operands, e.g. array transposition is required when we want to perform element-wise addition on row `R` and column `C` vectors.

In order to perform array transposition in the runtime, we could encode it as a higher-order function accepting as arguments the addition function (`plus`) with its operands (`R` and `C`). Then, inside this higher-order function, we test if operands are compatible (`size(R) == size(C)`; we assume both variables are numerical) and either returns the original expression (`R+C`) or an expression with compatible operands (`R+C'`). However, in all these cases, the dynamic checks are encoded inside user-defined functions, e.g. `d_slice`, which prevent the JIT compiler from merging instructions together (see Section 4.4.2). Therefore, there exists a performance trade-off between the benefit of not performing static code analysis and the interference with JIT compiler capabilities to compile instructions together.

5.1.2 Eliminating redundant 0-initialisation

In Chapter 4, while building the execution model, we have noticed that array slicing generates not only data copy but also an allocation pattern before the copy (with a large amount of memory store operations). Our initial guess was that the built-in array slicing in MATLAB initialises the allocated memory

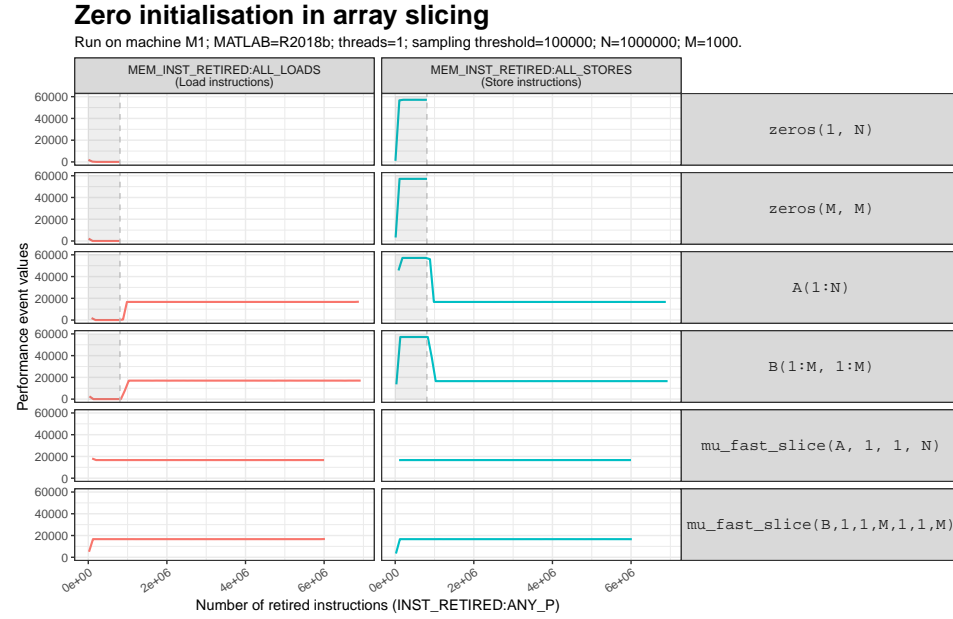


Figure 5.1: Array slicing performs 0-initialisation (marked as a grey region) before copying the extracted elements. The initialisation is an execution region with a high amount of store instructions.

before making a copy to conform with “first-touch” policy. In other words, array slicing performs a redundant 0-initialisation by writing zeros to the memory and then overwrites these zeros with the an actual copy of data.

To better visualise the problem, we have prepared performance profiles of: (1) explicit memory allocation for one and two dimensional data with memory allocation function `zeros`, and (2) explicit array slicing `A(1:end)`, `B(1:end, 1:end)` depicted in Figure 5.1. At the beginning of these codes, the 0-initialisation pattern occurs in execution regions consisting of a high amount of store instructions and a lack of load instructions. At this point, it is highly probable that the store instructions write zeros to the memory.

Listing 5.2: The `mu_fast_slice()` function is a faster alternative to the built-in array slicing. `mu_fast_slice()` removes redundant memory initialisation before the data copy with replacement of `mxMalloc()` by `mxMalloc()`.

```

1 void mexFunction(int nlhs, mxArray *plhs[], int nrhs, ...)
2 {
3     auto *array = (double *)mxGetData(prhs[0]);
4     ...
5     // Allocation without 0-initialisation
6     mxSetData(plhs[0], mxMalloc(sizeof(double) * num_elems));
7     double *output = (double *)mxGetData(plhs[0]);
8 
```

```

9  // Copy one dimensional data
10 for (size_t k_out = 0, k_in = lower_bound; k_in <= upper_bound; k_in
    += step, k_out++) {
11     output[k_out] = array[k_in - 1]; // Data copy
12 }
13 }

```

In order to test our hypothesis, we have implemented in C (MEX API) a new indexing function `mu_fast_slice()` depicted in parts in Listing 5.2. The `mu_fast_slice()` takes the same arguments as `d_slice()` and it does not perform explicit 0-initialisation, because we have replaced MATLAB function `mxCalloc()` by `mxMalloc()` which only reserves a memory. In normal circumstances, the memory initialisation is important because it forces an explicit memory allocation (“first touch”). Therefore, at this point, we should stay cautious because the `mu_fast_slice()` changes how MATLAB performs a fundamental operation, the array slicing.

While investigating reasons for removing the 0-initialisation, we have considered the concept of *optimistic memory allocation* in Linux systems (flag `vm.overcommit_memory`). The `malloc()` function in Linux only reserves the memory for allocation, but the function never accesses the memory. Therefore, the reserved memory which was not accessed for a long time, might be taken by another process when needed. In this context, the explicit memory initialisation is desired as it touches the memory and cements the memory reservation. Nevertheless, the *optimistic memory allocation* is not a problem in MATLAB for two reasons: (1) both `mxCalloc()` and `mxMalloc()` use `malloc()` internally (we have validated this claim by analysing calls to the C runtime library `glibc` in the machine code with the Intel VTune [99]), but more importantly, (2) `mu_fast_slice()` access the memory explicitly just right after the allocation by performing data copy. Hence, the use of `mxMalloc()` with immediate data copy mimics the `mxCalloc()` and it renders the 0-initialisation obsolete.

In Figure 5.1, the fifth and sixth rows present the performance of the optimised array slicing with the `mu_fast_slice()` function for one and two dimensional arrays. The two dimensional version of `mu_fast_slice()` accepts three more arguments for the range expression of the second dimension. The new array slicing executes fewer instructions than the built-in array slicing; however, the performance gain highly depends on the size of the array slice. The bigger the slice, the more costly 0-initialisation is and the bigger the performance gain is from removing the initialisation. Furthermore, the presented transformation can be systematic and applied everywhere for two reasons: (1) it is always legal; and (2) it does not require any profiling beforehand.

5.2 Repacking of array slices

Array slicing affects the performance of MATLAB programs in two ways: (1) it creates a copy of the requested subset of an array; and (2) it sometimes prevents the JIT compiler from merging two or more operations into one instruction block. Usually, array slicing is necessary for programs and we need to perform the copy at some point. However, it is possible to move the copy before all associated computation. Therefore, the computation uses only references to the already copied array slices which allow the JIT compiler to merge and execute operations together. In this section, we show how to extract array slices and better schedule instructions by repacking array slices into new variables.

Listing 5.3: Repacking of array slices on `crni3` loop from the LCPC16 benchmark suite.

```

1 % Original vectorised code
2 X(1:(N-1)) = (B(1:(N-1)) - C(1:(N-1)) .* X(2:N)) ./ D(1:(N-1));
3 % After repacking of array slices
4 tmp_b = B(1:(N-1));
5 tmp_c = C(1:(N-1));
6 tmp_x = X(2:N);
7 tmp_d = D(1:(N-1));
8 X(1:(N-1)) = (tmp_b - tmp_c .* tmp_x) ./ tmp_d;
```

Transformation. Listing 5.3 presents the simple idea behind the repacking of array slices. The transformation replaces every indexed read reference (array slice) in the right-hand side of the assignment (line 2) by a reference to a temporary variable `tmp_*` which holds the array slice (line 8). The code on lines from 4 to 7 depicts how array slices are copied into new temporary variables `tmp_*`. The idea is somehow analogous to *packing* data into vector registers where the temporary variables act as our registers [139] or *packing* sparse subset of data into dense variables just for the computation [140]. However, the metaphor ends because temporary variables act only as aliases of the array slices. In other words, we are just *repacking* the slices; hence, the name.

Application. The transformation is especially useful in cases where array slicing prevents the JIT compiler from executing operations together (Chapter 4). Repacking replaces array slices in the statement with references which allow the JIT compiler to merge and execute operations in a single instruction block. For finding beneficial applications of the repacking, we propose to use our execution model for expressions from Chapter 4. The model allows predicting the order of instructions in the expression before and after the repacking. Furthermore, the model works directly from the source

code of the program and requires no prior execution (static model). In other words, our model is a suitable candidate for a transformation heuristics for source-to-source compilers.

The knowledge about the order of instructions after the repacking indicates if the repacking creates new instruction blocks, execution regions where many operations execute together. In general, the repacking yields two results: (1) no change to the order of instructions; or (2) creation of new instruction blocks by merging other blocks. In case (1), the repacking does not improve, nor deteriorate the program performance. However, the sole existence of new instruction blocks (2) is not sufficient to guarantee the performance improvement. Other conditions including the size of input data, the number of threads, and the version of JIT compiler (MATLAB version), among others have to be taken into account.

Results. Figure 5.2 presents the relative increase of the performance after the repacking of array slices measured on three kernels: `crni3` loop from LCPC16 [1], `state_fragment` (kernel 7) from Livermore ¹, and `s211` from TSVC [141, 142] benchmark suites.

A recurrent pattern in the data, that we can observe in Figure 5.2, is a better performance of the repacking obtained on the newer version of MATLAB R2018b. A possible explanation is the improved working of the JIT compiler. In this case, the repacking reveals a massive opportunity for JIT compilation, which the results show.

The repacking for TSVC/s211 kernel decreases the performance for the majority of tests. However, the execution model from Chapter 4 can predict this outcome. The result of the model is the same instruction chain of the code before and after the repacking. In other words, the repacking is not profitable because it does not create any new instruction block with operations executed together.

Other two loops LCLP16/crni3 and Livermore/state_kernel are perfect examples of how the performance gain from the repacking depends on the size of input data, the number of threads, the MATLAB version, and the code itself. In the current form, the execution model from Chapter 4 is insufficient to answer if the repacking increases the performance. Thus, the repacking should be considered with the code profiling to find out if the transformation is beneficial in the given context.

5.3 Range simplification

Ranges created with the colon operator, e.g. `1:2:N`, `colon(50, -1, 25)` are an integral part of almost every MATLAB program. The colon operator creates a new vector or extracts a subset of array elements in array slicing.

¹<https://www.netlib.org/benchmark/livermore>

Performance change after repacking of array slices

Run on machine M2.

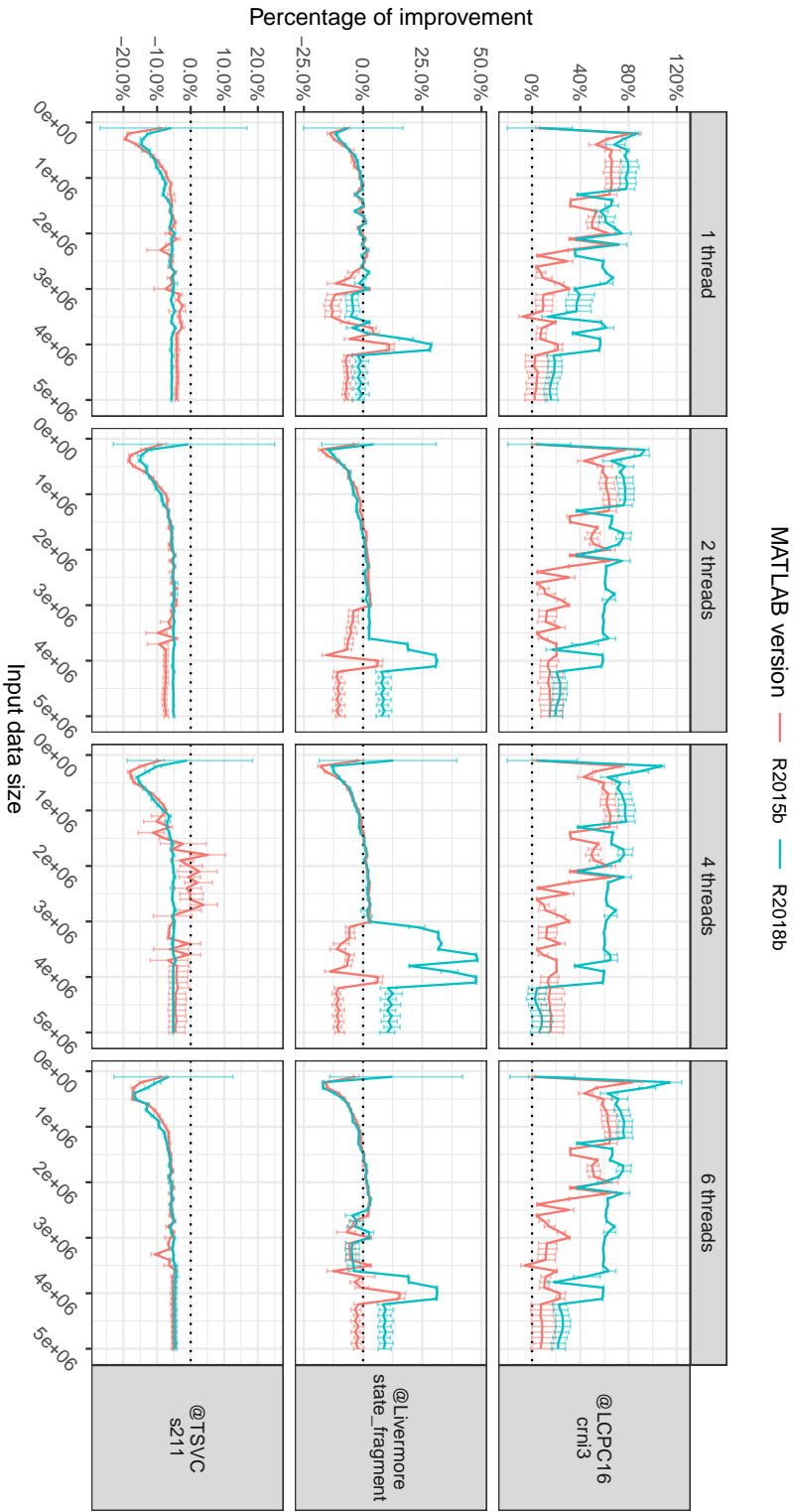


Figure 5.2: Results of applying repacking of array slices. Not every computation benefits from the transformation (TSVC/s211).

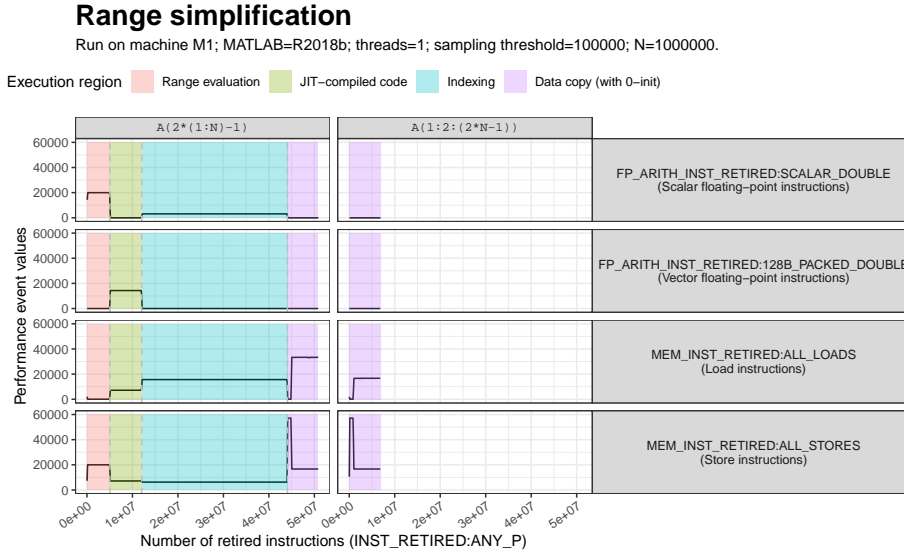


Figure 5.3: Array slice with an arithmetic expression as index generates complex execution regions (on the left). However, an array slice with a range expression results only in the **data copy** (on the right).

Moreover, MATLAB optimises the use of the colon operator in the context of array slicing.

Figure 5.3 presents performance event profiles for two array slices, one created with an arithmetic expression $A(2*(1:N)-1)$ and the other with just one range $A(1:2:(2*N-1))$. The figure depicts profiles of four performance events which count the number of instructions: scalar and vector arithmetic instructions, load and store instructions, from top to bottom on the figure. Both codes are equivalent; they create the same range and extract the same elements of an array (odd elements). However, the first code is an expression with multiplication, subtraction, and range $1:N$. Therefore, the expression requires evaluation with three distinctive steps before performing the data copy of the required array elements. Those steps represent the following execution regions: **range evaluation** of $1:N$; multiplication and subtraction compiled into one instruction block with the JIT compiler (**JIT-compiled code**); and indexing of array elements from the computed indices (**indexing**). Moreover, the same performance profiles occur for array slicing with indices stored in an external variable or when the indices are a random permutation of values from 1 to N . In short, this first version of the code is highly inefficient.

The second code $A(1:2:(2*N-1))$ skips all three steps and moves directly to the **data copy**. The colon operator acts here as a recipe for the JIT compiler on how to extract requested array elements. The ranges created with the colon operator are simple arithmetic progressions. Thus, the JIT compiler can use the ranges in the generated code to directly access array elements

during computations. However, the JIT compiler is not able to transform explicit arithmetic expression like in the case of the first code.

Transformation. To simplify the arithmetic expression $A(2*(1:N)-1)$, we frame it as an affine function $f(k) = \alpha k + \beta$. We consider here only expressions in the form of an affine function, where α and β are expressions which evaluate to scalar values. If k is a range, then $f(k)$ is a new range where every element is multiplied by α and increased (or decreased) by β . However, because k has the form of `[from:step:to]`, we can use the function $f(k)$ to rewrite the k to the final shape as follows `[f(from): α *step:f(to)]`.

5.4 Profile-guided loop vectorisation

For many years, MATLAB programs expressed with array operations were faster than those with for-loops. Therefore, in order to accelerate MATLAB programs, researchers have applied automatic *loop vectorisation* to MATLAB [1, 9, 21], a transformation of for-loops to array operations previously known for Fortran [71, 140, 143]. However, in recent years, the Just-In-Time (JIT) compiler in MATLAB has improved the performance of not only array operations but also for-loops resulting in a dilemma about which for-loops should be vectorised [1, 31].

Chen et al. well illustrate the dilemma in their work about `Mc2Mc`², an automatic vectorising compiler for MATLAB [1]. The `Mc2Mc` has no heuristics for selecting loops which could profit from vectorisation, and as a result, `Mc2Mc` vectorises every for-loop systematically, if only the vectorisation is valid for the loop. Moreover, with the systematic vectorisation, the profit from several profitable loop vectorisations could be balanced out by the performance decrease from other vectorised loops.

The result of Chen et al. work [1] indicates the systematic vectorisation is beneficial for 8 out of 9 benchmarks in a purely interpreted execution with average speedups $\times 19.1$ for Octave 4.0 and $\times 7.65$ for MATLAB R2013a. However, the systematic vectorisation with newer versions of MATLAB with the JIT compilation enabled (version \geq R2015b) often decreases the performance with an average speedup of $\times 1.02$ for MATLAB R2013a and a slowdown of $\times 0.77$ for MATLAB R2015b. With the JIT compilation, the minority of benchmarks benefit from the systematic vectorisation: only 5 on R2013a and 2 on R2015b.

In this section, we explore *profile-guided vectorisation* as one of the solutions to the dilemma of loop vectorisation, which was proposed by Chen et al. [1]. The technique creates an interval of input data size for which the loop vectorisation is profitable. Later, during program execution, the interval

²<https://github.com/Sable/Mc2Mc>

Table 5.1: Experiment reproduction of systematic vectorisation from the paper by Chen et al. [1]. Although, the exact speedups differ, the reproduction follows the same patterns of speedups and slowdowns as in the original work.

Benchmark	Description	Speedup	
		Chen [1]	Us [31]
<i>backprop</i>	Backpropagation algorithm	0.71	0.81
<i>bs</i>	Black-Scholes model	15.0	8.33
<i>capr</i>	Gauss-Seidel method	0.79	0.85
<i>crni</i>	Crank-Nicholson method	0.83	0.81
<i>fft</i>	Fast Fourier Transform	0.59	0.64
<i>nw</i>	Needleman-Wunsch algorithm	0.96	1.00
<i>pagerank</i>	PageRank	0.94	0.94
<i>mc</i>	Monte Carlo simulation	2.02	2.22
<i>spmv</i>	Sparse Matrix-Vector Multiplication	0.013	0.02

is used as a condition which selects the faster version of the code: loop or vector operations.

Experiment reproduction. In this section, we have reproduced the results from the paper by Chen et al. [1] on our machine M1 (full specification in Table A.1). We have followed the authors’ instructions by computing speedups from average execution time after 5 repeated measurements. The reproduction focuses on MATLAB R2015b with enabled JIT compiler, because for this MATLAB version loop vectorisation results in a considerable slowdown.

Table 5.1 depicts performance speedups of loop vectorisation from the work of Chen et al. [1] and our reproduction. Although the results differ, we observe the same speedups and slowdowns for particular benchmarks. The result validates that loop vectorisation in MATLAB is responsible for the performance change and not particular to machine configuration. The result also shows that 7 out of 9 benchmarks (grey rows) are slower after the systematic vectorisation. Therefore, our further analysis focuses solely on the 7 benchmarks in a need for a better vectorisation policy than the systematic approach.

Code generation improvement. The Mc2Mc compiler creates, for each vectorised loop in the generated code, a dedicated variable which stores the iteration space of the loop, mostly a range expression. Array slicing uses later that variable in the generated code. In Section 5.3, we have established that array slicing with indices stored in a variable performs prolonged indexing process, before making a copy. However, inlining the range expression into

Performance of loop vectorisation

Run on machine M1; MATLAB=R2015b; threads=2.

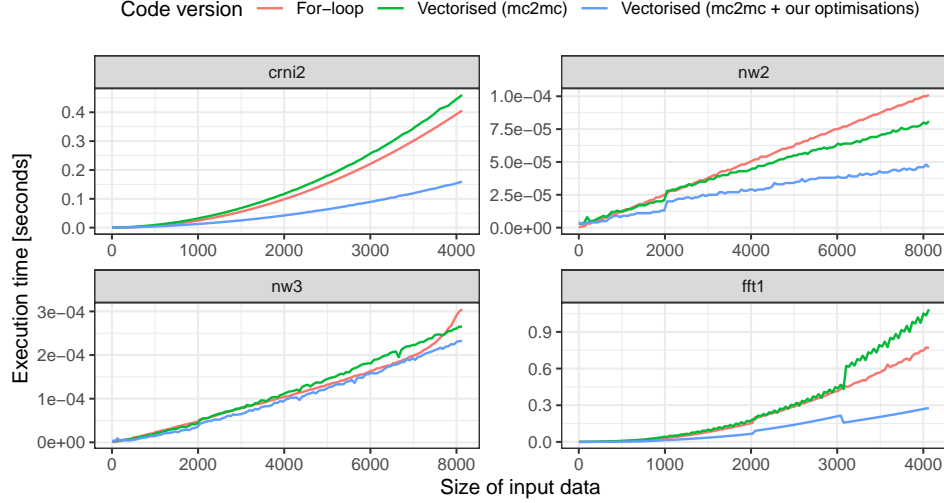


Figure 5.4: The execution time of loops `crni2`, `nw2`, `nw3`, `fft1` in original (*for-loop*) and vectorised versions; without and with our optimisations. Source code of the loops is located in Figure D.1 inside Appendix D.

the array slice omits the indexing and performs the copy directly. Moreover, `Mc2Mc` does not perform range simplification, as described in Section 5.3.

Figure 5.4 presents how the range expression inlining and the range simplification increase the performance of the loop vectorisation. Vectorised code with our optimisations performs as fast, or faster than the original loop. The results are especially important in all cases where non-optimised vector codes generated by the `Mc2Mc` are slower than loops. Figure D.1 in Appendix D depicts the differences between tested loops and their versions.

Loop profiling. After improving the code generated by the `Mc2Mc`, we investigate how sensitive the performance of loop vectorisation is to the change of input size data. We have assumed the input size is synonymous to the number of loop iterations because in our context more loop iterations require more data to traverse over and the opposite.

For the profiling task, we have extracted and profiled 17 loops from 7 benchmarks, which have shown performance slowdown after the systematic loop vectorisation (see Table 5.1). The profiling code randomly initialises arrays used in the computations to a range of sizes from 1 to 3 200 000 with various steps, depending on the loop, but it always includes the input size used in the study of Chen et al. [1]. This approach is valid assuming the loop has no control dependence [144] which is the case for our loops. Next,

Table 5.2: Results of loop profiling (on MATLAB R2015b) depicts the interval of profitable data size when the loop vectorisation improves the performance of the loop. \emptyset indicates the loop is never optimised by vectorisation in the profiled range, which always includes values of experiment data size from the work of Chen et al. [1]. Intervals marked in grey intersect with experiment data size, thus, increasing the performance of corresponding loops.

Loop	Experiment data size	Profitable data size	
		Mc2Mc	Mc2Mc+opt
backprop1	{17, 2850001}	\emptyset	≥ 255
backprop2	2	≥ 4033	≥ 257
backprop3	{17, 2850001}	\emptyset	≥ 385
backprop4	2	\emptyset	≥ 257
capr1	8	≥ 20	≥ 17
capr2	20	≥ 3329	≥ 385
capr3	49	≥ 5953	≥ 321
crni1	2300	≥ 161	≥ 193
crni2	2300	\emptyset	≥ 289
crni3	2300	\emptyset	≥ 1217
fft1	256	\emptyset	≥ 417
fft2	2, 4, 8...256	\emptyset	≥ 129
nw1	4097	\emptyset	≥ 65
nw2	4097	≥ 1665	≥ 257
nw3	4097	≥ 7681	≥ 193
pagerank1	1000	\emptyset	≥ 273
spmv1	{2, 3}	≥ 6337	≥ 321

the code measures the execution time between 10 to 100 times for each loop, depending on the loop complexity. Finally, we take *minimal* execution time from the measurements to obtain better stability and discards first 1 to 10 measurements which usually show an overhead of a warmup stage of MATLAB execution engine [145].

Table 5.2 depicts intervals of input data size for which the loop vectorisation is profitable, without (Mc2Mc) or with our optimisations (Mc2Mc+opt). Data size marked as \emptyset indicates the loop does not benefit from vectorisation for the profiled range. If we compute the intersection of the profitable data sizes with sizes observed in the experiment by Chen et al. [1], we find which loops benefit from vectorisation. For the non-optimised code (Mc2Mc), vectorisation improves only two loops *crni1* and *nw2*, marked in grey. Our optimisations, range simplification and range expression inlining, improve further performance of all loops by shifting to the left intervals of profitable data size. For example, applying our optimisations to the non-optimised code

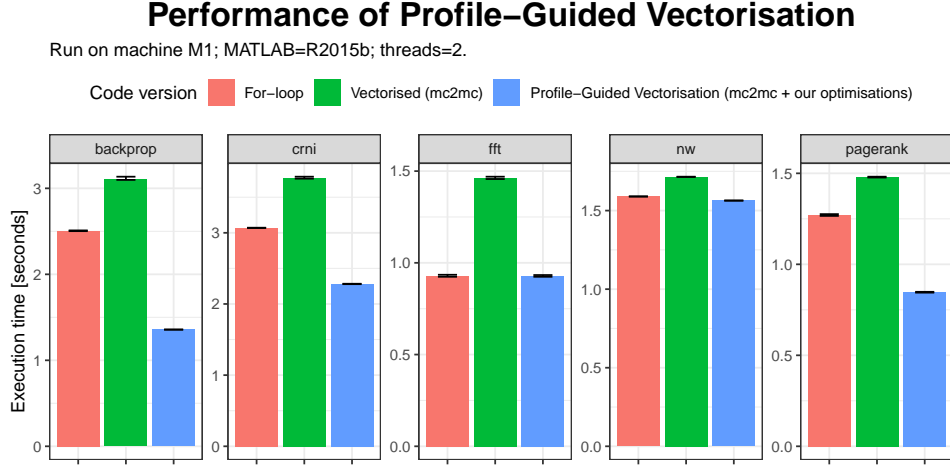


Figure 5.5: Profile-guided vectorisation obtains faster or equal execution in comparison to the loop code. More importantly, the profile-guided approach does not degrade performance as opposed to standard, systematic vectorisation in `Mc2Mc` compiler.

`nw2`, decreases the starting point of beneficial vectorisation from 1665 to 257. In other words, the optimised vector code is faster than the original loop `nw2` even for smaller number of iterations than for the pure `Mc2Mc` version. Moreover, now, up to 10 loops from the work of Chen et al. [1] can benefit from the optimised vectorisation, because their data sizes used in the experiment intersect with intervals of profitable data size.

In order to integrate the profile-guided vectorisation, our compiler prepares two code versions: original loop and vectorised code. Moreover, the compiler inserts `if` statement to check if the number of loop iterations is inside the interval of profitable data size (Table 5.2). If that is the case, then the program runs the vector code. The runtime checks are essential especially for loops `backprop1`, `backprop3`, and `fft2` because for them, not every experiment data size belongs to the profitable data size.

The presented profile-guided vectorisation has a considerable advantage for two reasons: (1) the profiling examines loops for a whole range of possible input data sizes preparing it for the future reuse; and (2) the profiling result is integrated into the generated code, thus, allowing the code to easily choose different code versions when values of variables and input data change in the program. Therefore, the approach facilitates program optimisations without hurting MATLAB property of rapid prototyping. However, at the price of running the loop and the vector code multiple times.

Conclusions. A profile-guided vectorisation is a viable approach to the selection of candidates for loop vectorisation in MATLAB programs. The

profiling requires only one parameter the number of loop iterations, thus, significantly simplifying the procedure of finding the conditions when a loop is worth vectorising. With the improvements to *array slicing* in the code generated by the Mc2Mc compiler, we have significantly improved performance of three benchmarks: `crni` by 25.7%, `pagerank` by 33.3%, and `backprop` by 45.8% as depicted in Figure 5.5.

Benchmarks and results presented in this section are available on-line³.

5.5 Conclusion

This chapter presents the results of fusing performance event profiles (PEP) from Chapter 3 with information about program execution from Chapter 4. Without the performance profiles, we could not discover how MATLAB performs data copy during array slicing. Moreover, without the profiles, we could not analyse and formalise the program execution in MATLAB which lead us to concepts of the *instruction block* and the *instruction tree*. All these components allowed us to make the following contributions:

- Introduction of *dynamic array slicing* which tests, during the program execution, if the indexing expression covers the whole array (see Section 5.1.1). As a result, this function removes redundant data copy without performing any static code analysis. This approach of deferring certain tasks to the runtime applies to other operations like array transposition or data replication.
- Reimplementation of MATLAB built-in array slicing with `mu_fast_slice` for one and two dimensional arrays. The new array slicing removes redundant zero initialisation improving performance of the operation (see Section 5.1.2).
- Introduction of *repacking of array slices* (presented in Section 5.2), a transformation which extracts array slices from MATLAB expressions and inserts them into new variables. Thus, allowing JIT compiler to merge operations from the expression into a single *instruction block*.
- Introduction of a simple, yet, powerful *range simplification* transformation which converts expressions with ranges, e.g. `3*(1:N)+2`, to pure ranges `5:3:(3*N+2)` (see Section 5.3). The transformation benefit comes from the fact that array slicing with pure ranges skips the evaluation of the complex expression and indexing process seen in Figure 5.3.
- Analysis of *profile-guided loop vectorisation* applied to MATLAB programs described in Section 5.4 and [31]. First proposed by Chen et

³<https://github.com/quepas/array2018-profile-based-vectorization>

al. [1], the profiling of loops is a valid technique for precise application of vectorisation to MATLAB loops. In our work, we have combined the vectorisation with *range simplification* to find even more optimisation opportunities for loops which benefit from vectorisation only at a specific interval of loop iterations obtained from the profiling.

Chapter 6

HU!M compiler

Résumé

Dans ce chapitre, nous présentons notre compilateur HurryUp!MATLAB qui implémente le modèle d'exécution et les transformations de code présentées dans les sections précédentes. L'outil est un compilateur source-à-source qui prend en entrée le code MATLAB et produit du code MATLAB modifié avec des performances identiques ou supérieures. Le compilateur effectue plusieurs analyses bien établies telles que l'analyse du flot de contrôle, l'analyse du flot de données et l'analyse des dépendances de données de la boucle. Plus important encore, HU!M est un compilateur entièrement automatique, et il ne nécessite aucune information de code supplémentaire.

Introduction

Compilers not only translate programming languages to a machine code which is understandable and executable by computers, they also translate from one language into another, or like in our case, compilers restructure and transform an existing code without changing the underlying programming language. The latter compilers are called *source-to-source compilers* (or *transpilers*) with examples of ROSE [146], PIPS [147], Cetus [148, 149], or Mc2Mc [1]. The most common usage of the transpilers is to increase program performance by transforming into a more efficient form or by deploying it on another (performant) architecture, e.g. parallel, multi-core, many-core, GPGPU.

In this chapter, we present our prototype source-to-source compiler HurryUp!MATLAB (HU!M) dedicated to increase performance of MATLAB/Octave programs on multi-core processors. Throughout our work, we have used HU!M to prototype new code transformations, perform quantitative analysis of the code, and prepare the codes for experiments.

6.1 Overview

HU!M is a source-to-source compiler which takes an input MATLAB/Octave program and performs two things: (1) analyses the input code; and (2) transforms the code to achieve better performance. The analysis part (1) consists of mainly standard compiler analyses such as: *control-flow analysis* [150], *data-flow analysis* [151], and *loop dependence analysis* [152], which give a fundamental information about the existing relationships between the statements inside programs.

The analysis stage contains also MATLAB-specific (or array programming specific) *dimensionality analysis* by Birkbeck et al. [21], which reasons about the size of arrays changing over the course of the program execution. The knowledge about changing arrays and their shape is necessary for loop vectorisation [1, 9]. Moreover, within the compiler, we have implemented our *execution model for MATLAB expressions* as explained in Chapter 4 as a part of the analysis stage. The model predicts how MATLAB divides the program into instruction blocks, generated by the Just-In-Time (JIT) compiler. With the complete prediction of instruction blocks, the compiler can apply code transformations which reorder, fuse, or divide expressions and statements in order to obtain an optimal performance (or at least close to).

The transformation part is a collection of separate code transformations: classical automatic *loop vectorisation* [71, 72]; and two of our code transformations *fast array slicing*, which replaces built-in array slicing routine with our optimised C MEX reimplementation, and *repacking of array slices*, which extracts array slices and replaces them by references, thus, creating a single instruction block for the computation.

The HU!M compiler is written in Java because of the language simplicity and educational values (easy to read, modify, and extend). With the choice of Java, we clearly state the prototype nature of our compiler. Moreover, the compiler only targets the stand-alone MATLAB/Octave without additional toolboxes in order to limit the scope of the analyses and the transformations. Furthermore, with the compiler, we only target the optimisation of MATLAB/Octave code executed on multi-core processors (MATLAB can also perform computations on GPU/CUDA and distributed architectures with the use of `Parallel Computing Toolbox`).

In the next section, we briefly present other compilers, which have influenced the design and implementation of ours. Furthermore, we decompose and describe in details the analysis and the transformation stages of our HU!M source-to-source compiler.

6.2 Influences

The development of **HU!M** was influenced by several existing open-source compilers. From these projects, we have collected and reused a few features and approaches, most suitable for our purpose — a code transformation of MATLAB/Octave programs.

LLVM. LLVM is a compilation toolchain designed for building new compilers and language environments [75]. Today, LLVM is a basis for **clang** (a C/C++ compiler), runtime environments for Swift, Rust, and many research projects, e.g. Polly — implementation of loop transformations based on the polyhedral model [153].

In LLVM, every analysis and code transformation is designed around passes. Single pass traverses a portion of code, e.g. basic blocks or for-loops, to perform code analysis or transformation. While code analyses only gather information about the code, code transformations alter the code which might require its re-analysis. In **HU!M**, we use the concept of the pass to create four distinctive passes: **IRAnalysis**, **Analysis**, **IRTransformation**, and **Transformation**.

The first two passes **IRAnalysis** and **Analysis** both perform analysis, but on different inputs. The **IRAnalysis** works on intermediate representations (IR) which are homogeneous, e.g. control-flow graph (CFG) consisting of basic blocks and data-dependence graph (DDG) containing statements. Whereas, **Analysis** in **HU!M** works on the whole or only parts of the AST, which is heterogeneous [154]. In other words, the AST consists of distinct nodes and **Analysis** can work on, e.g. any AST node, only **SingleAssignment**, or on all top-level **ForLoop** in a MATLAB script. The same distinction is true between **IRTransformation** and **Transformation**.

The passes also differ in their inputs and outputs. The analysis passes have only one input parameter: an IR or the AST. The outputs of those passes are accessed through predefined methods (getters) because single analysis might output many distinctive information. Transformation passes take only specific input and output. For example, **LoopVectorisation** is a transformation pass which takes a **ForLoop** node and vectorise it into a **Region** node — a linear chunk of code. Furthermore, the transformation pass **IRTransformation** is used in between code analyses. For example, **BuildCFG** pass converts an input AST into a CFG.

Halide. Image processing with stencil computations raises challenges to efficient code generation that **Halide** tries to tackle. **Halide** is a state-of-the-art language and compiler which expresses and treats separately what is computed (*algorithm*) with how it is computed (*schedule*) [155]. However, what influenced our compiler was something unrelated to improving code generation for image kernels. Instead, we have reused the implementation

of the visitor for mutating an AST. The altering of the AST is conditional because it only modifies the AST root node if any of its subtrees have changed. Listing 6.1 presents an example of `visitPlus` method from our compiler, which mutates the left and right AST subtrees and recreates the *Plus* node only if the subtrees have changed.

Listing 6.1: Conditional AST mutation implemented as a visitor pattern.

```

1 Node Mutator::visitPlus(Plus plus) {
2     Node left = mutate(plus.getLeft());
3     Node right = mutate(plus.getRight());
4     if (left.sameAs(plus.getLeft()) && right.sameAs(plus.getRight())) {
5         return plus; // If the subtrees haven't change, return the original node
6     }
7     return new Plus(left, right); // ... otherwise return the modified node
8 }

```

Without checking if subtrees have changed, altering a tree with the visitor pattern recreates every node, even if not mutated. However, the approach presented in Listing 6.1 transforms only the necessary parts of the input AST. Therefore, the `Mutator` invalidates the results of code analysis only on modified parts of the AST leaving intact other results of the analysis.

McLAB. McLAB [14, 156] is an umbrella project of tools dedicated to the analysis, compilation, and transformation of MATLAB code. The universe contains static analysers (McSAF [77], Tamer [70], McFLAT [157]), compilers (AspectMatlab [158], MiX10 [16], Mc2For [15]), and Just-In-Time (JIT) compiler (McVM [22]). However, for us, the most interesting part consists of static analyser: McSAF, Tamer, and Tamer+. In these project, we have analysed the data-flow solver, value propagation in Tamer, and many other components. Moreover, we have reused some part of the MATLAB grammar defined in the fronted of the McLAB (which actually works on a subset of MATLAB). In the end, we did not decide to use the McLAB in our study, because our focus has switch into the performance analysis over time. Moreover, our analyses and code transformations are fairly simple, requiring only a straightforward code base.

Cetus. This source-to-source compiler is a comprehensive solution to the analysis and transformation of C language [148, 149]. The compiler contains a simple implementation of the data-flow solver with reaching definition analysis. Moreover, Cetus is implemented in Java, just like our compiler. Therefore, we have reused several classes related to the def-use chains and data-flow graphs which are universal across the analysis of imperative languages. Furthermore, Cetus contains working implementations of loop data dependence tests which was of great help to our efforts in implementing this analysis.

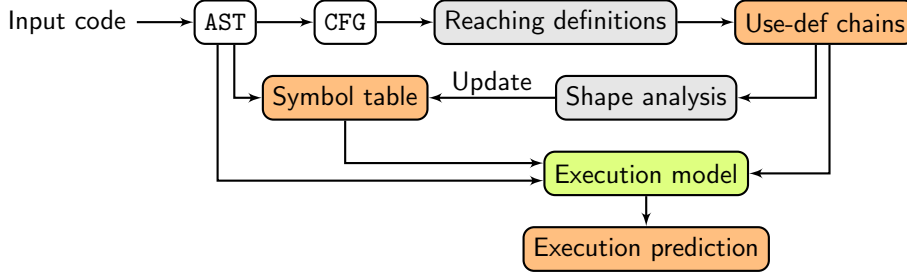


Figure 6.1: Overview of the analyses implemented in HU!M.

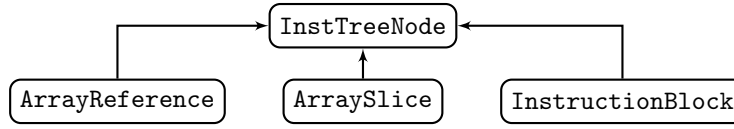


Figure 6.2: Hierarchy of classes representing the instruction tree.

6.3 Code analysis




HU!M compiler implements several well-known analyses which are crucial for the automatic analysis and transformation of MATLAB/Octave programs. Figure 6.1 presents an overview of the most important analyses for our work, some of them are already mentioned in the previous section. In the compiler, we work only on abstract-syntax tree (AST) representation which is well-suited for code transformations and simple analyses.

From the description, we omit the implementation of *shape analysis* by Birkbeck et al. [21] and loop dependence analysis based on Integer-Linear Programming (ILP) formulation, because both of them are fairly straightforward and known. Instead, we focus on the implementation of our execution model from Chapter 4.

Execution model

HU!M compiler implements the execution model described in Chapter 4. The implementation is straightforward because the compiler already works on the abstract-syntax tree (AST) code representation. Therefore, we skip the first step of the model — the conversion from a MATLAB expression to the AST (in Section 4.5.1). Instead, we start with a procedure for building an instruction tree from the AST. The full implementation of the execution model consists of two classes: `BuildInstructionTree` and `ExprExecutionModel`.

Building instruction tree. The first class, `BuildInstructionTree`, performs a translation from an input AST to a corresponding instruction tree. The instruction tree is represented as a hierarchy of four classes depicted

in Figure 6.2. All three subclasses: `InstructionBlock`, `ArrayReference`, and `ArraySlice` directly correspond to nodes  *instruction block*,  *array reference*,  *array slice* from Section 4.5. Moreover, only `InstructionBlock` holds children nodes, because both `ArrayReference` and `ArraySlice` are always used as leaves.

The transformation from AST to instruction tree uses the visitor pattern to traverse an AST in a post-order. Every time the procedure encounters an array reference or slice, it issues a corresponding node. However, if the procedure encounters an instruction, it not only creates a node for the instruction block, but it also connects the already visited arguments with this block as its children.

Listing 6.2: Definition of `InstructionBlock` and `Instruction` classes.

```

1 public InstructionBlock extends InstTreeNode {
2     private Multiset<Instruction> instructions = HashMultiset.create();
3     private Set<ArrayReference> references = Sets.newHashSet();
4     private Set<ArraySlice> slices = Sets.newHashSet();
5     ...
6 }
7
8 public Instruction {
9     private String opcode;
10    private CompilationMode compilationMode; // {SINGLE, COMBINABLE, UDF}
11    ...
12 }

```

Every node of the instruction tree holds additional information. Both, `ArrayReference` and `ArraySlice` store a reference to the AST node `ReadRef` which represents the name of the accessed array with indexing expressions. Moreover, `InstructionBlock` holds information described in Section 4.2: (1) instructions inside the block; (2) set of array references used in the computation; and (3) set of array slices accessed from inside the block.

Listing 6.2 shows the definition of the `InstructionBlock` and `Instruction` classes. A single instruction represented as `Instruction` contains its name (`opcode`) and the compilation mode (`compilationMode`), which is a property expressing how the JIT compiler will use the instruction (execute alone or merge with other instructions). Section 4.4 describes the compilation mode in details.

Obtaining minimal instruction tree. `ExprExecutionModel` class directly uses the instruction tree to perform three steps from Section 4.5.1: (1) remove array references and store them inside instruction blocks; (2) create minimal instruction tree presented in Algorithm 2; and (3) flatten the minimal instruction tree to obtain the instruction chain which indicates what execution regions MATLAB will create to execute an input expression.

Listing 6.3: Building of the minimal instruction tree. The code follows Algorithm 2.

```

1  public class ExprExecutionModel implements ExecutionModel {
2      ...
3      public boolean canMerge(InstTreeNode node) {
4          return node instanceof InstructionBlock && onlyCombinableFunctions(node);
5      }
6
7      public boolean hasRightSibling(InstTreeNode node) {
8          InstTreeNode parent = node.getParent();
9          int indexOfTheChild = parent.getChildren().indexOf(node);
10         return indexOfTheChild < parent.getChildren().size()-1;
11     }
12
13     public void buildMinimalTree(InstTreeNode node) {
14         List<InstTreeNode> revChildren =
15             Lists.newArrayList(Lists.reverse(node.getChildren()));
16         for (Iterator<InstTreeNode> it = revChildren.iterator(); it.hasNext();) {
17             buildMinimalTree(it.next());
18         }
19
20         if (canMerge(node) && canMerge(node.getParent())) {
21             if (!hasRightSibling(node)) {
22                 node.getParent().mergeWith((InstructionBlock) node);
23                 node.getChildren().forEach(child -> node.getParent().addChild(child));
24                 node.getParent().getChildren().remove(node);
25             }
26         }
27     }
28     ...
29 }

```

Stage (1), removing of array reference nodes, is a straightforward visitor pattern where children of each instruction block are analysed. In the case where a child is an array reference, the child node is inserted into the set of **references** and then detached from the parent.

The next stage (2), reduction of an instruction tree to the minimal instruction tree form, is a direct implementation of Algorithm 2 depicted in Listing 6.3. The method traverses the instruction tree in a post-order manner with right-to-left visiting of children (lines 14–18). Then, the algorithm checks if a visited node can merge with its parent (lines 20–21). If the condition is true, the node merges with its parent and it is removed from the tree entirely (lines 22–24). The result of this procedure is a minimal instruction tree where every pair of instruction blocks that could be merged was merged.

Finally, stage (3) translates the minimal instruction tree into an instruction chain which is a flatten representation of this tree. In Section 4.5.2, we have showed how the instruction tree predicts data copy and computation region occurring during the execution of MATLAB expression (Figure 4.9). These predictions indicate, e.g. which version of an expression has less instruction

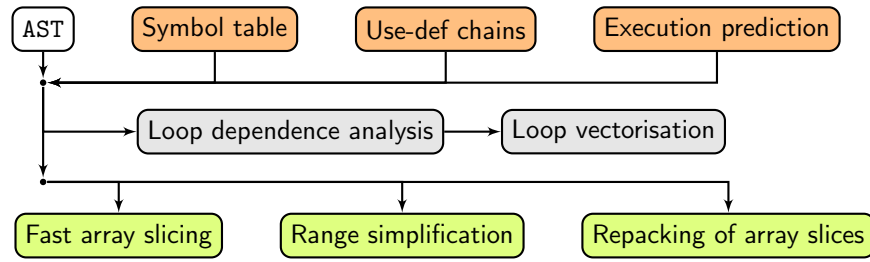


Figure 6.3: Overview of transformations in HU!M.

blocks, or which expression after reordering will have less instruction blocks and which one has more.

6.4 Code transformation

In this section, we present three code transformations implemented in HU!M. Figure 6.3 shows how these transformations use information coming from code analyses described in Section 6.3. All presented transformations are implemented as `Transformation` passes.

6.4.1 Loop vectorisation

With the emergence of vector computers, programming languages like Fortran, have received new instructions designed especially for vector computations [71, 140]. The vector form of programs (also called vectorised code) is available in MATLAB as well in a form of array operations. Loops in MATLAB performs scalar operations, however, vectorised code often uses vector instructions (from ISA extensions: MMX, SSE, AVX on Intel architecture). Moreover, as shown even fairly recently by Chen et al. [1] and in our work [31] (described in Section 5.4), loop vectorisation is still a viable optimisation technique for MATLAB programs.

Our implementation of loop vectorisation is based on techniques and on the *theorem of dependence* described by Kennedy and Allen [71, 72] which are also used by other MATLAB source-to-source compilers [1, 21]. HU!M contains `LoopVectorisation` transformation pass implementing a simplified algorithm `vectorize` from the book by Allen and Kennedy [72] (Figure 2.1 in this book).

Analysing data dependences. Our procedure starts by building a data dependence graph (DDG) for the analysed loop nest [152]. For this task, we use a loop dependence analysis based on Integer Linear Programming (ILP) described in Section 6.3. The DDG represents relationships between elements of arrays which are read from and write to in the same iteration (loop

independent dependence) or across many iterations (loop carried dependence). Those dependences between elements may prevent instructions from being vectorised directly, because some computations depend on previous ones, therefore, these computations cannot execute in parallel.

Finding vectorisable statements. The data dependence graph shows which statements depend on each other. The next step is to find in the DDG strongly connected components (we use Kosaraju’s algorithm for this task [159]) which indicate cycles of dependent statements that cannot be vectorised. Further, the DDG is translated into a π -graph where each cycle of statements or single statement translates to a single node (called π -node) [71]. Then, the π -graph is topologically sorted to obtain a legal execution order of instructions. Finally, our algorithm generates vectorised code for each π -node containing single statement. Other π -nodes with cyclic statements are wrapped around a for-loop and executed sequentially as initially.

Listing 6.4: Loop vectorisation often requires performing array transposition.

```

1 A = zeros(1, n);
2 B = rand(1, n);
3 C = rand(n, 1);
4 % Loop version does not care about the layout of arrays
5 for i=1:n
6     A(i) = B(i) + C(i);
7 end
8 % Vectorised version with the transposition of column vector C
9 A(1:n) = B(1:n) + C(1:n).';

```

Transforming array layout. During the vectorisation of a given statement, we use the dimensionality analysis by Birkbeck et al. [21] (Section 6.3) to validate if all arrays after the transformation are compatible. A common case when they are not compatible, is when a vectorised element-wise expression contains a mix of column and row vectors as depicted in Listing 6.4. In this case, the dimensionality analysis marks which vectors should be transposed to make arrays compatible again, such as vector `C` in Listing 6.4 which must be transposed (`.'`) so that the addition (+) preserves its semantics. Otherwise, without the transposition, the addition will return an error due to the incompatible sizes in MATLAB until the version R2016b. In R2016b and after, the addition would perform implicit array expansion of both vector operands into matrices of the same size by replicating the vectors and performing element-wise addition on the new matrices.

Dynamic code selection. HUI!M also generates *if-else* statement for dynamic code selection of instructions which performance depends on variables

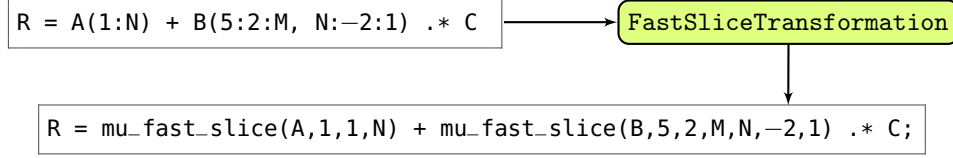


Figure 6.4: Overview of the fast array slicing substitution in HU!M.

in programs. In Section 5.4, we have showed that loop vectorisation is beneficial only if the loop performs a specific amount of iterations p (or work on sufficiently large data of size p). If p value, obtained from loop profiling, is provided to `LoopVectorisation`, then the transformation generates a code for selection between the loop or its vectorised version by simply comparing the profitable data size p with the number of loop iterations l available during program execution. In other words, if $l \geq p$ then the vectorised code is selected and executed.

6.4.2 Fast array slicing substitution

In Section 5.1, we have presented function `mu_fast_slice` which implements fast array slicing by entirely bypassing the built-in indexing routine `subsref` in MATLAB. The function is a part of `MatlUp`, our C MEX library consisting of reimplemented and optimised MATLAB constructs like the array slicing.

The function comes in two variants: for 1D arrays `mu_fast_slice(array, from, step, to)`; and for 2D arrays `mu_fast_slice(array, r_from, r_step, r_to, c_from, c_step, c_to)` where prefix `r` indicates a slice for rows and `c` for columns. Both variants of the function skip unnecessary stage of 0-initialisation which occurs for small and medium slices (under ≈ 15 MB), as explained in Section 5.1. However, the other benefit of switching from the built-in array slicing to `mu_fast_slice` comes from the fact that the translation can be performed automatically using our compiler HU!M.

In the compiler, transformation pass `FastSliceTransformation` implements the substitution of array slicing as depicted in Figure 6.4. The implementation is based on the `Mutator` class which visits the AST tree in post-order and replaces only modified sub-trees to minimise the number of reinitialised nodes. In our case, we only replace `ReadRef` nodes representing array slice in an expression. The conditions for `ReadRef` to be replaced are: (1) it references an array; (2) it contains an indexing part; (3) the indexing expression is a range. The (3) part is especially important, because arguments of the `mu_fast_slice` function for 1D arrays consist of the reference to an array slice and three numeric arguments `from`, `step`, and `to`, which express the range from the indexing expression. For higher dimensions, the `mu_fast_slice` takes additional set of `from`, `step`, `to` arguments for the higher

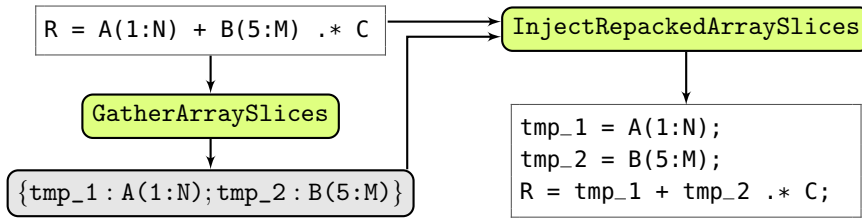


Figure 6.5: Overview of the repacking of array slices implemented in HU!M.

dimension.

Moreover, the same code transformation as for `mu_fast_slice` can be applied to `d_slice` function from Section 5.1. The `d_slice` reimplements array indexing by dynamically checking if the indexing expression is redundant. If the indexing expression includes all array elements, then the explicit array slice is redundant, and `d_slice` returns directly the reference to the array.

6.4.3 Repacking of array slices

The repacking of array slices is a code transformation which replaces array slices by array references in a vectorised expression. The benefit of this transformation is the removal of array slices which prevent the MATLAB JIT compiler from merging instruction blocks in this expression. Chapter 4 describes how instruction blocks are formed and what prevents them from merging. In Chapter 5, we already detailed the repacking, thus, in this section we focus only on the implementation.

The implementation of the repacking of array slices consists of the main transformation pass `RepackingArraySlices` and two stages encapsulated in classes: `GatherArraySlices` and `InjectRepackedArraySlices`. Figure 6.5 shows how an input expression is transformed to a repacked form during the two transformation stages.

The first class `GatherArraySlices` collects the array slices which could be extracted and packed into new, temporary variables `tmp_*`. The second class `InjectRepackedArraySlices` replaces gathered slices by references to newly created temporary variables `tmp_*`, which hold the original array slices. The output code of the transformation is an expression which uses only array references and as a result, the code has equal or smaller minimal instruction tree than before the transformation.

6.5 Conclusion

In this chapter, we have introduced and described HU!M — our source-to-source compiler extensively used during our work on accelerating MATLAB/Octave

programs. Although, the compiler is only a prototype, it is a good basis for further investigations of code analysis and transformation techniques. The simplicity of the compiler design and the choice of Java as its primary programming language, makes it a perfect tool for educational purposes too.

HU!M consists of several well-known code analysis and transformation techniques: dimensionality abstraction [21], loop dependence analysis [152], or loop vectorisation [72]. Moreover, in the compiler, we have implemented our own techniques described in the previous chapters: execution model for MATLAB expressions (Chapter 4), fast array slicing (Section 5.1) and repacking of array slices (Section 5.2).

In the future, we plan to extend the scope of available code analyses and transformations in the compiler. Especially, we would like to focus more on automatic loop transformations: loop tiling [27], scalar replacement [85], and loop unrolling [160]. The reason is that part of our unpublished work showed promising results of applying these transformations to MATLAB/Octave loops too. Moreover, a natural extension for our execution model is the consideration of array types. For this reason, we plan to implement a type analysis scheme similar to the work of De Rose and Padua on FALCON compiler [48] with typing aspects by Hendren [161].

Chapter 7

Evaluation of the execution model and code transformations

Résumé

L'évaluation comprend l'analyse du modèle d'exécution présenté et deux transformations de code : le reconditionnement des tableaux et la simplification de la plage d'indexation. Pour le modèle d'exécution, nous avons testé la capacité du modèle à prédire l'exécution de l'expression MATLAB. Les résultats montrent une grande précision du modèle. De plus, le modèle permet de guider diverses transformations simples de l'expression, ce qui augmente leurs performances. Ces transformations comprennent la division de l'expression en sous-expressions pour forcer les opérations de tableau sur les registres vectoriels et le réarrangement des opérations commutatives afin de regrouper les opérations sous un nombre plus petit de blocs d'instructions.

Le temps d'exécution des programmes après la simplification des plages d'indexation diminue toujours. En outre, le gain de la transformation varie de 227.9% à 1011.7% dans divers contextes. D'autre part, les avantages du reconditionnement des sections de tableaux sont plus nuancés. Bien que la transformation puisse augmenter les performances du code, même d'un facteur deux, elle peut également les diminuer jusqu'à 20%. Dans sa forme actuelle, notre modèle d'exécution est incapable de prédire le résultat du reconditionnement.

Introduction

This chapter describes the evaluation of three items: (1) the execution model from Chapter 4; (2) range simplification from Section 5.3; and (3) repacking of array slices from Section 5.2. Previously, we have shown that our model can predict the order and type of execution regions seen during the program execution. Now, Section 7.1 demonstrates how well it works on more MATLAB expressions and how it can be used to improve the program performance. Moreover, Section 7.2 quantifies the gain in execution time of

applying range simplification. Finally, Section 7.3 shows the benefits and pitfalls of using repacking of array slices.

7.1 Evaluation of the execution model

Table 4.4 in Chapter 4 presents eight MATLAB expressions with their minimal execution trees obtained using our model. In this section, we execute these expressions to validate if our model correctly predicts their execution. Moreover, we show what happens when we modify some of these expressions by reordering operands of commutative operations (e.g. addition `plus`, element-wise multiplication `times`) or by splitting an expression into several sub-expressions.

Evaluation setup. Our evaluation uses two machines described in Table A.1 with processors implementing different microarchitectures, *Skylake* and *Coffee Lake* introduced in 2015 and 2017, respectively. Although their microarchitectures are different, the set of native performance events is the same on both of them (171 events). Therefore, the evaluation and plotting of diagrams are considerably simplified, as we can run the same tests on both machines, and prepare uniform diagrams.

The evaluation is performed on MATLAB in versions R2015b (the first version with the LXE execution engine) and R2018b. These versions have the same, current architecture of the interpreter and the JIT compiler, even though there is a 3-year gap between these versions. Nevertheless, evaluation of them could show the broad applicability of our execution model.

7.1.1 Model precision

Every minimal instruction tree (MIT) from Table 4.4 precisely predict the order and type of execution regions. Figure 7.6 and Figure 7.1 contain the measured execution regions for expressions corresponding to MIT 1 and 2. Execution regions predicted by MIT-3 to 8 are presented in Figures D.3 to D.8 in Appendix D. Furthermore, during the work on our execution model, we have tested it with more than 200 expressions without finding any counterexamples.

7.1.2 Splitting expressions

Expressions MIT-2, 3, 4 and 5 perform scalar operations when they could perform vector operations. Figure 7.1 depicts the execution of the expression of MIT-2. This expression has four instruction blocks, where the second block performs the sub-expression `fix(B.*C)`. This sub-expression executes only scalar operations, although, element-wise multiplication `.*` usually performs vector instructions. The reason behind this behaviour is that the `fix` function

forces scalar arithmetic on the rest of the instructions from the block. Other combinable functions from Table 4.3 which force scalar arithmetic are `asinh`, `atan`, `atan2`, `ceil`, `cos`, `cosh`, `exp`, `fix`, `floor`, `mod`, `pow2`, `rem`, `round`, `sin`, `sinh`, `tan`, `tanh`. Each time we mix vector instructions inside an instruction block with one of the above functions, the code executes in scalar mode.



Figure 7.1: Expression `floor(A)+sqrt(fix(B.*C))` has four instruction blocks with mixed scalar and vector arithmetic.

In order to avoid performing scalar operations by vector instructions, we propose to split the expression into several sub-expression to keep scalar operations together in separate instruction blocks. To illustrate how to split expressions, consider expression `floor(A) + sin(fix(B .* C))` of MIT-3 from Table 4.4. This code contains only combinable functions; thus, it requires one instruction block for the whole expression as depicted in Figure D.3. Since the expression `floor(A) + sin(fix(B .* C))` has one instruction block, every arithmetic operations in this block perform scalar computation. Instructions `floor`, `sin` and `fix` force scalar mode of computations when mixed with other functions. Therefore, our goal is to keep these three instructions separated from addition (+) and element-wise multiplication (.*).

Figure 7.2 presents how to split the input expression into four sub-expressions. Only two functions + and .* perform vector operations; thus, we split them in a way that they do not mix with scalar functions — `tmp_mul` variable stores element-wise multiplication. Furthermore, both operands of the addition have scalar operations; therefore, we store them in `tmp_left` and `tmp_right` variables.

Figure 7.3 depicts the program execution after the splitting of the MIT-3 expression. After the splitting, there are four instruction blocks instead of one. Two of them perform vector operations of + and .* operators. However,

```

1 % Minimal Instruction Tree #3 (MIT-3)
2 R = floor(A) + sin(fix(B .* C))
3 % Splitting MIT-3
4 tmp_left = floor(A);
5 tmp_mul = B .* C;
6 tmp_right = sin(fix(tmp_mul));
7 R = tmp_left + tmp_right;

```

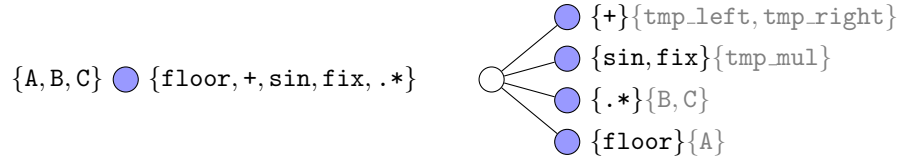


Figure 7.2: MIT-3 expression splitted into several sub-expression with their minimal instruction trees depicted below.

the splitting of the expression fragments our computation. Moreover, `+` and `.*` take only a small fraction of the computation (`sin`, `floor` and `fix` are much more compute-intensive). Therefore, even with vector operations, the expression performs in 0.0457 and 0.0467 seconds before and after the splitting, respectively. In order to obtain performance gain from splitting, vector operations in the expression must take a considerable amount of computation. Moreover, the splitting introduces more instruction blocks, each with its execution overhead.

7.1.3 Reordering operations

Expressions MIT-2, 6 and 7 have commutative functions, and their operands can be reordered. Consider expression `log(A) + B + C(1:N)` of MIT-6 (depicted in Table 4.4) which has an array slice to variable `C` in the right subtree. This slice prevents two additions `+` from merging together. However, by knowingly modifying and reordering the expression, we can alter its instruction tree in a way which changes the number and the content of instruction blocks without changing the semantics of the computation.

On the top-level, expression `log(A) + B + C(1:N)` consists of commutative addition (we can change the order of operands) and associative (we can change the order of evaluation). In our example, if we swap the reference to array `B` with the array slice `C`, we make sure that none of the right-sibling operations blocks the rest of the expression in the instruction tree (see Algorithm 2 in Section 4.5.1 for more information). Therefore, by modifying the expression to `log(A) + C(1:N) + B`, we obtain a simpler instruction tree depicted in Figure 7.4. Furthermore, Figure 7.5 presents the execution regions of the expression before and after the reordering. The mean execution time is 0.0224 and 0.0186 seconds before and after reordering, respectively, which significant

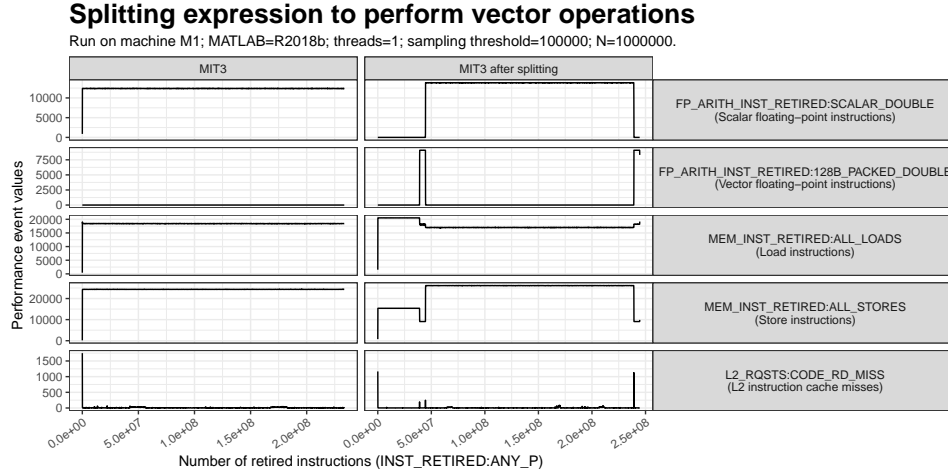


Figure 7.3: Some functions prevent others from performing vector operations. Splitting an expression into sub-expressions might allow performing vector operations at the cost of the increased number of instruction blocks.

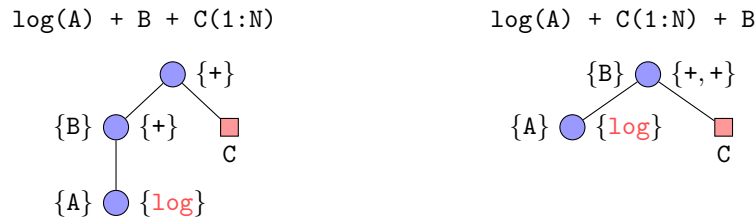


Figure 7.4: Minimal instruction trees for the expression before and after reordering. The reordered expression has fewer instruction blocks.

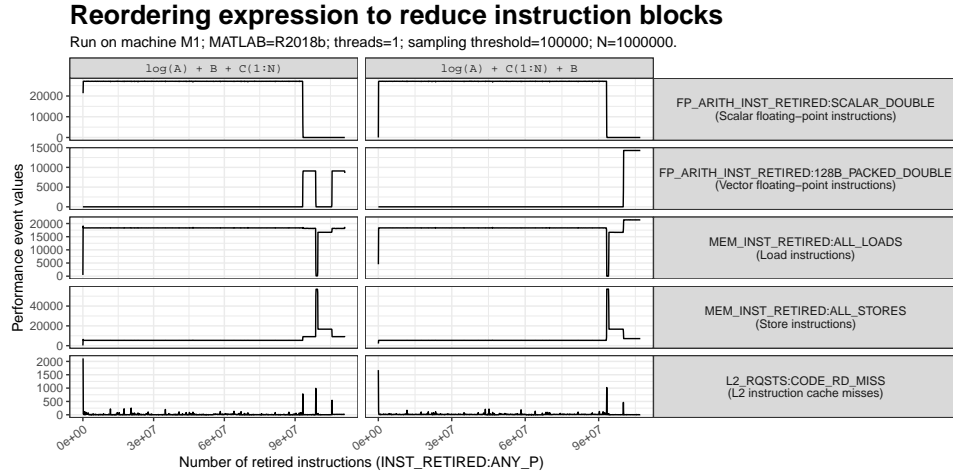


Figure 7.5: Reordering of the expression to reduce the number of instruction blocks.

performance improvement for this example.

7.1.4 Information limit of performance profiles

In this section, we present a short discussion about the limits of information encoded with performance event profiles (PEP), used to discover and build our execution model. The discussion is closely related to the evaluation of the model, more precisely, to the mapping of execution regions to particular machine instructions.

Expression of MIT-1 from Table 4.4 is a composition of two functions `sum(round(A))`. In this composition, the only combinable function is `round`, which can coexist with other MATLAB functions in the same instruction block (see Table 4.3). Therefore, each function requires its instruction block which is seen in Figure 7.6.

Figure 7.6 depicts two execution regions corresponding to two instruction blocks, one for `round` and the other for `sum`, respectively. However, only one of them performs floating-point operations. The first execution region lacks arithmetic operations because the family of `FP_ARITH_INST_RETIRED:*` events count only specific machine instructions listed in Table 7.1. The list includes operations like `ADD`, `SUB`, `MUL`, `MAX`, `SQRT`, but it lacks rounding `x86` operations, e.g. `roundpd`, `roundsd`. However, even with the ability to count rounding `x86` operations, it is not guaranteed that MATLAB JIT compiler generates these instructions for the `round` built-in function.

In order to check which `x86` instructions the MATLAB JIT compiler generates for the `round` function, we can use Intel PIN [120] or Intel VTune [99] (which is also based on the PIN). PIN has many already prepared tools (so-

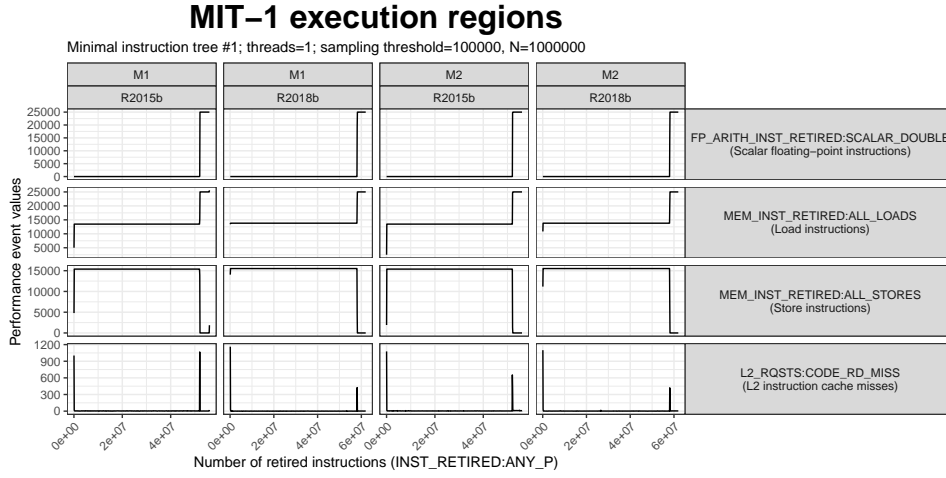


Figure 7.6: Expression `sum(round(A))` has two instruction blocks.

called *PIN-tools*) which scan, analyse, or modify the machine code of the running program. For our purpose, the `opcodemix` tool counts the number occurring `x86` instructions. However, this approach counts them globally without consideration of basic blocks. On the other hand, Intel VTune while counting the `x86` instructions, it can profile the code and indicates which part of the program is executed the most (in terms of basic blocks). Therefore, if we test compute-intensive programs, it is apparent which basic block is responsible for the bulk of operations, executing the content of MATLAB built-in functions.

The results of our analysis show that MATLAB `round` function is not using any of `x86 round[sp][sd]` instructions. Instead, the “round to the nearest decimal or integer” of the `round` function is implemented using arithmetic right shift SAR instruction, along with others. Therefore, when analysing the occurrence of instructions and building our model with performance event profiles (PEP), we need to consider not only what performance events measure, but also what kind of instructions MATLAB generates, and how we can measure them.

7.2 Evaluation of the range simplification

In MATLAB, operator `colon (:)` defines a range of values with a specified start point, endpoint, and interval between two given values in this range. The range expresses an arithmetic progression. Section 5.3 shows that using ranges as indices in array slicing (`A(1:2:(2*N-1))`) reduces the amount of computation in comparison to using expressions which perform element-wise operations on ranges (`A(2*(1:N)-1)`). Therefore, in Section 5.3, we have introduced *range simplification* which transforms such expressions to an equivalent range.

Table 7.1: Machine instructions counted by the `FP_ARITH_INST_RETIRED:*` family of performance events on Skylake and Coffee Lake microarchitectures.

Operation	x86 instruction	Description
ADD	<code>vadd[sp][sd]</code>	Add floating-point values
SUB	<code>vsub[sp][sd]</code>	Subtract floating-point values
MUL	<code>vmul[sp][sd]</code>	Multiply floating-point values
DIV	<code>vdiv[sp][sd]</code>	Divide floating-point values
MIN	<code>vmin[sp][sd]</code>	Minimum of floating-point values
MAX	<code>vmax[sp][sd]</code>	Maximum floating-point values
RCP	<code>vrcp[sp]s</code>	Compute reciprocals of single-precision floating-point values
RSQRT	<code>vrsqrt[sp]s</code>	Compute reciprocals of square roots of single-precision floating-point values
SQRT	<code>vsqrt[sp][sd]</code>	Square root of floating-point values
DPP	<code>vdp[sd]</code>	Dot product of packed floating-point values
FM(N)ADD/SUB	<code>vfm[add sub]*</code>	Fused multiply (negative) add/subtract of floating-point values

`[sp]` — scalar or packed.
`p` — only packed.
`[sd]` — single or double-precision.
`s` — only single-precision.

Listing 7.1 and Listing 7.2 show two code examples of indexing expressions transformed into simplified ranges. Although MATLAB programmers tend to avoid complex indexing expressions, automatic vectoriser `mc2mc` by Chen et al. [1] generates complex expressions and stores them in external variables.

The use of ranges in array slicing allows the MATLAB JIT compiler to generate a machine code which skips the explicit range evaluation, and instead, it uses the range values directly in the array indexing. Figure 7.8 depicts how the simplified range omits three execution regions responsible for (1) evaluating the range; (2) performing arithmetic operations on the range; and (3) indexing the array with just computed indices. In this section, we

Listing (7.1) Indexing all elements.

```

1 % Indexing expression
2 B = A((0:(N-1)) + 1);
3 % Range simplified
4 B = A(1:N);
5 % Variable with indices
6 I = 1:N;
7 B = A(I);

```

Listing (7.2) Indexing even elements.

```

1 % Indexing expression
2 B = A((1:2:(N-1)) + 1);
3 % Range simplified
4 B = A(2:2:N);
5 % Variable with indices
6 I = 2:2:N;
7 B = A(I);

```

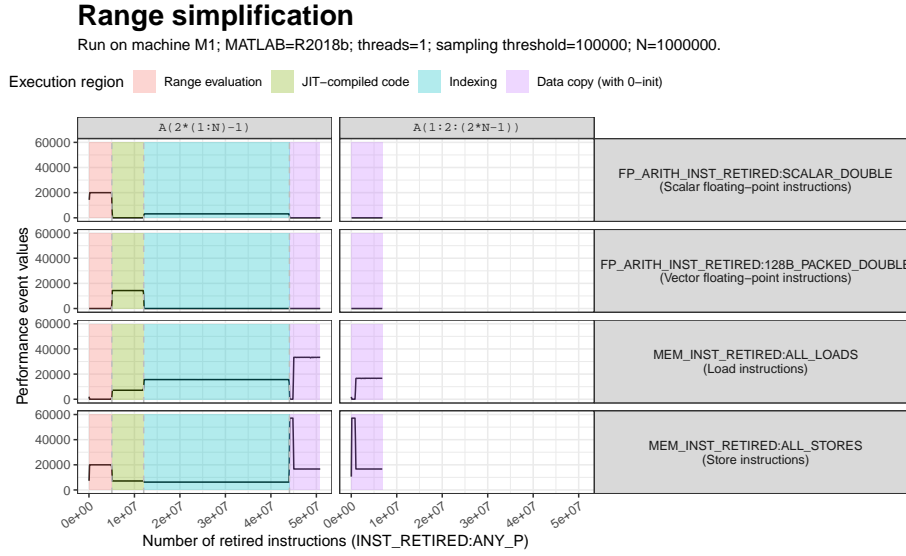


Figure 7.8: Array slice with an arithmetic expression as index generates complex execution regions (on the left). However, an array slice with a range expression results only in the **data copy** (on the right).

evaluate how omitting these execution regions affects the execution time of MATLAB codes. For this purpose, we have prepared two codes which perform indexing of all and even elements depicted in Listing 7.1 and Listing 7.2.

Execution time. Figure 7.9 presents the execution time of our code examples (from Listing 7.1 and Listing 7.2), executed on two machines, M1 and M2 using one thread. The results clearly show that range simplification massively reduces the execution time of array slicing with indexing expressions using ranges. The performance improvement spans from 306.5 % to 1011.7 % for indexing of all elements. Furthermore, for the indexing of even elements, the improvement ranges from 227.9 % to 769.3 %. Although array slicing takes less than 0.008 of a second for copying 5 000 000 elements (on machine M1), this cost increases proportionally to the number of array slices in the program. Therefore, the end benefit of using range simplification can be high for codes with many arrays.

Indices in variables. The code version *variable with indices* is interesting because it is almost a lower-bound for the execution time of indexing expression. The reason is simple; this code version performs the evaluation of the range and the indexing stage (depicted in Figure 7.8) just as the indexing expression does. However, the code lacks the arithmetic operation on the range $(\dots+1)$. In general, *variable with indices* executes faster or close to the performance of the indexing expression. This code version also shows that



Figure 7.9: Range simplification massively reduces execution time of array slicing with ranges.

indexing array with ranges is fast only when the range is used directly inside the slice and not stored in a variable (then, it requires prior evaluation).

7.3 Evaluation of the repacking of arrays

In Section 5.2, we have introduced a new code transformation called *repacking of array slices*. This transformation extracts array slices from an expression and puts them into temporal variables. Therefore, the transformed expression uses only references to the new variables. The benefit of repacking comes from the fact that array slices perform data copy, which is a distinctive execution region not combinable with other instructions. On the other hand, references to arrays do not generate any additional computation, and they can be easily used with combinable functions inside instruction blocks.

Our tests start with running 14 kernels from Livermore¹ and LCPC16 [1]. We execute each kernel on the machine M2 and two MATLAB versions R2015b and R2018b. Moreover, each kernel execution uses the size of input data spanning from 250 000 to 5 000 000 with the step of 250 000 elements. Figure 7.10 and Figure 7.11 presents the obtained results. Both figures indicate the number of repacked array slices by the transformation.

The exact rule when an array slice prevents merging instruction blocks is specified in Algorithm 2 in Chapter 4. However, the impact of repacking array slice is hard to quantify because it requires not only the knowledge

¹<https://www.netlib.org/benchmark/livermore>

Performance change after repacking of array slices

Run on machine M2 using Livermore kernels.

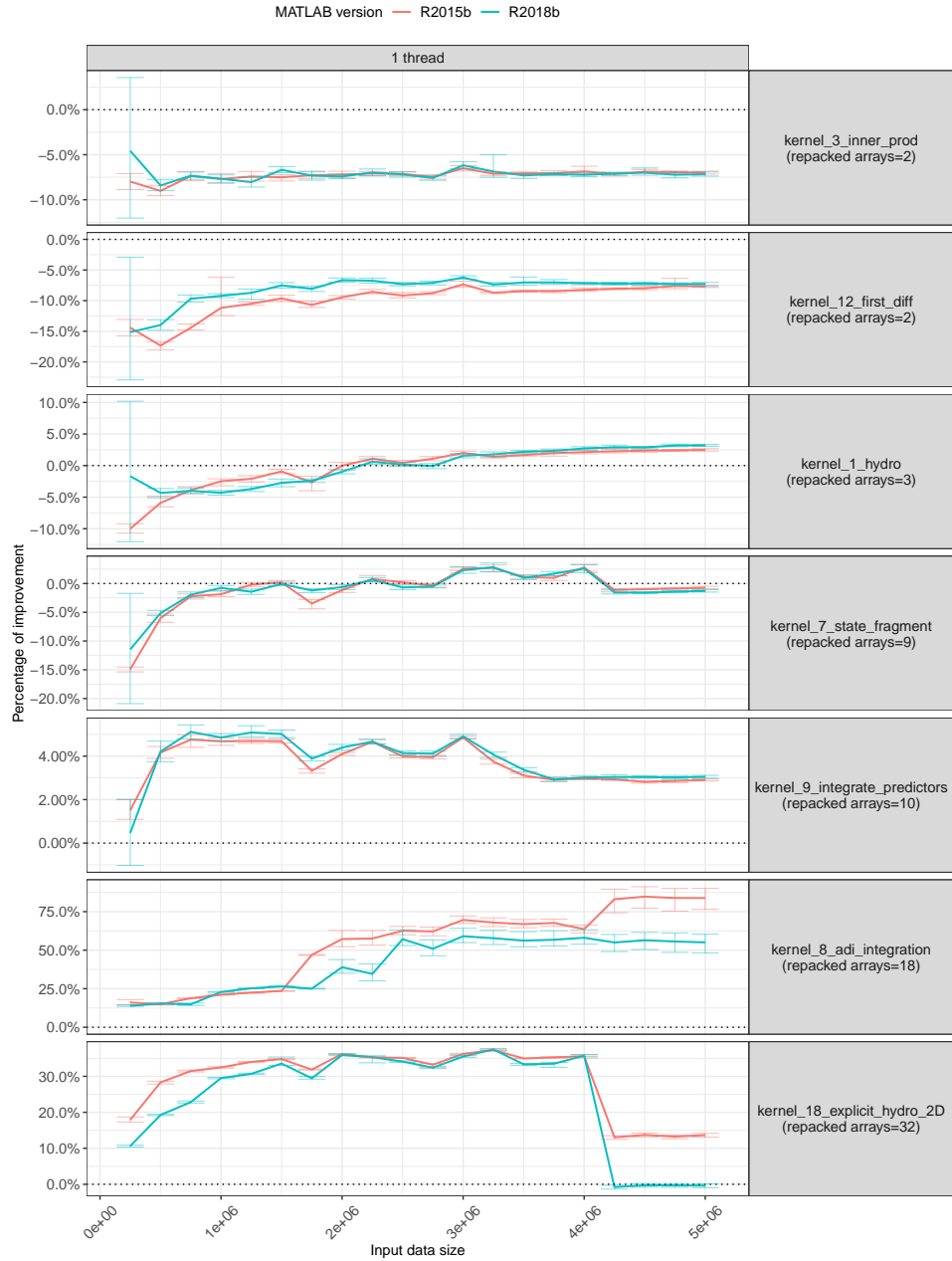


Figure 7.10: Repacking of array slices on Livermore kernels.

about instruction blocks and their order (which our execution model has), but also performance characteristics of particular functions, instruction blocks, and underlying execution regions. Without the knowledge, it is not possible to precise if the benefit from removing array copy and allowing the merge of two blocks will impact the overall performance beyond the measurement noise. Nevertheless, the results show consistent benefit in kernels with more than 10 repacked arrays (*integrate predictors*, *adi integration*, and *explicit hydro 2D* from Livermore suite). In other words, more repacked arrays create more opportunities for combining instruction blocks and increasing program performance.

Kernel with low number of repacked arrays, like kernel 3 from Livermore, `sum(z(1:LEN_1D) .* x(1:LEN_1D))`, creates no new optimisation opportunities. The `sum` function is non-combinable; therefore, the repacking cannot merge the element-wise multiplication with `sum`. Similarly, Livermore kernel number 12, `y(2:(LEN_1D + 1)) - y(1:LEN_1D)`, performs binary subtraction for which repacking is not creating any new instruction blocks (array slices always perform before this subtraction). Kernels from the LCPC16 suite experience the same problem, as the maximal number of the repacked arrays is 4 for the *crni3* kernel.

Obtained results also show two distinctive patterns. The first pattern is a “hill” of performance improvement occurring for the *backprop2*, *capr2*, and *capr3* LCPC16 kernels only on MATLAB R2015b (see Figure 7.11). The “hill” spans across 2 500 000 and 4 000 000 elements of input data (19.07 MB and 30.58 MB for a single array). These three kernels have one thing in common; they use `sum` function; however, the exact reason for this pattern requires further investigation.

The second pattern is a rapid decrease in performance after 4 000 000 elements which might be related to the changing cost of array slicing for this amount of data elements (consult Section 3.5 for more details) because of the benefit of using repacking which decreases with the decreasing cost of array slicing. Kernels *state fragment*, *explicit hydro 2D* from Livermore and *backprop2*, *backprop3*, *capr2*, *capr3*, *backprop1*, *crni3* from LCPC16 exhibit the second pattern.

The majority of these patterns occurs only in MATLAB R2015b which suggests a fundamental difference in how MATLAB environments execute particular functions.

7.4 Conclusion

In this chapter, we have evaluated our execution model for MATLAB expressions (from Chapter 4) along with two code transformations: range simplification (from Section 5.3) and repacking of array slices (from Section 5.2).

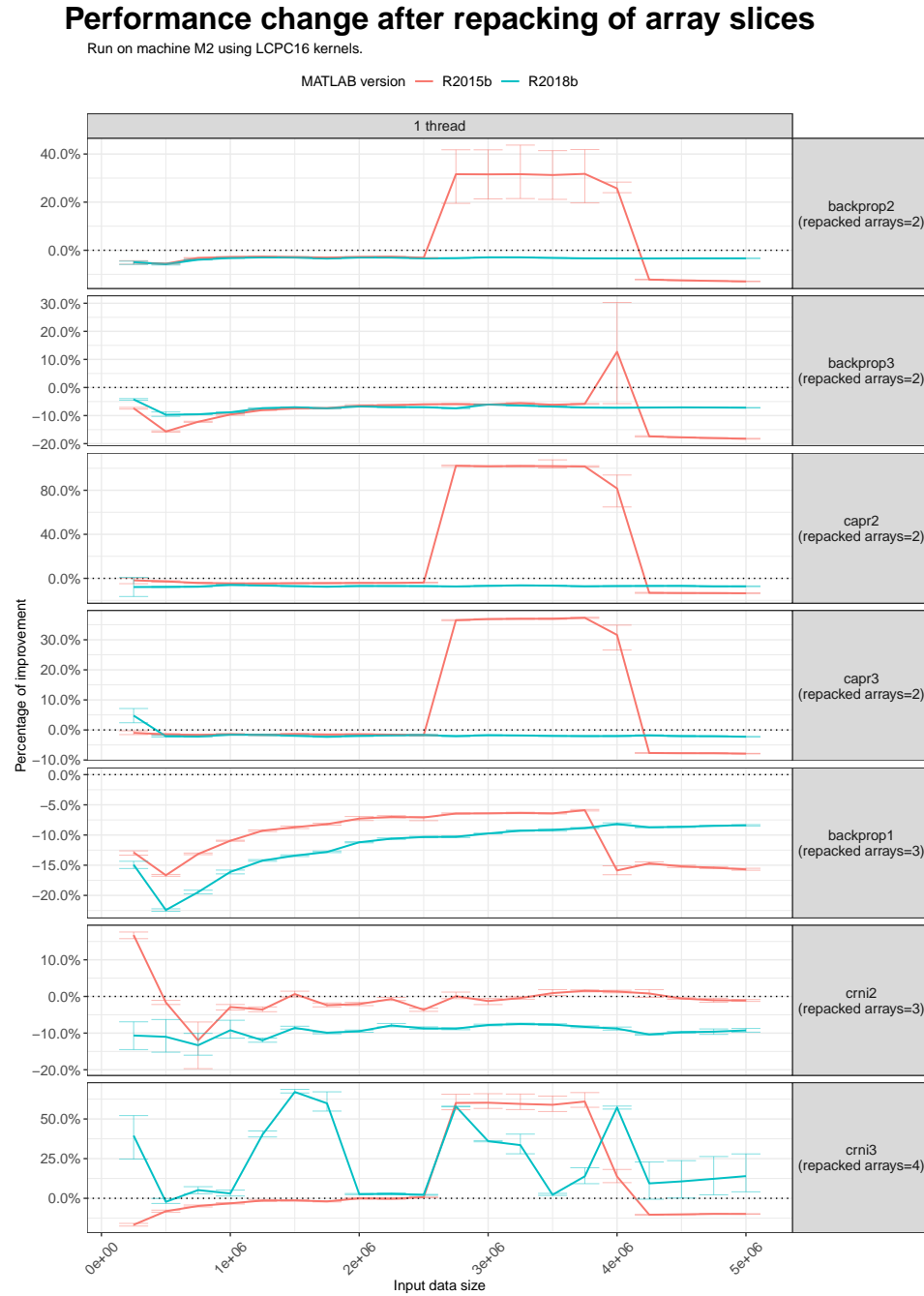


Figure 7.11: Repacking of array slices on LCPC16 kernels [1].

Execution model. The execution model for MATLAB expressions precisely predicts the order and type of execution regions occurring during the execution of example expressions from Table 4.4 in Chapter 4. Moreover, the model indicates which functions and parts of the expression execute in a single instruction block. Our test results have shown the broad applicability of the model which works for expressions executed on four experiment configurations with two versions of MATLAB R2015b and R2018b, and two machines M1 and M2 (consult Table A.1 for machine specifications).

Apart from testing the model precision, we have also highlighted several opportunities coming from the use of our execution model. To begin with, the model indicates instruction blocks which mix scalar and vector combinable functions. If an instruction block contains at least one scalar function, e.g. `ceil`, `cos`, `pow2`, then the whole block executes only scalar arithmetic operations, even for vector functions (this is clearly shown in Figure 7.3). The solution to having vector arithmetic operations in such an expression is to split it into sub-expressions which do not mix scalar and vector operations. Furthermore, reordering of expressions changes their minimal instruction trees; thus, changing the performance of expressions as well.

Range simplification. Section 7.2 presents the evaluation of range simplification transformation, which simplifies numerical ranges used as indices of arrays. The benefit from this transformation, on both machines M1 and M2, spans from 227.9% (for indexing even elements) to 1011.7% (for indexing all elements). Although this transformation is straightforward, it is also very powerful because ranges are recipes for the MATLAB JIT compiler on how to generate efficient code for array indexing. Moreover, in the array slicing context, pure ranges do not require prior evaluation.

Repacking of array slices. Finally, Section 7.3 demonstrates more examples of evaluation of the repacking of array slices. The repacking replaces array slices with references to them. This transformation changes the content and the number of instruction blocks created from an expression. Depending on these changes, the benefit of the repacking varies from decreasing the performance by -22% (kernel *backprop1* in Figure 7.11) to increase the performance by even 100% (kernel *capr2* in Figure 7.11).

In general, the results show that the benefit from the repacking increases with the number of repacked arrays. For the Livermore suite, three kernels with at least 10 repacked arrays (*integrate predictors*, *adi integration*, and *explicit hydro 2D*) are always faster after repacking. However, in the current form, our execution model is not able to predict the benefit of the repacking because it does not consider the effects of data size on the performance which profoundly impacts the performance results.

Chapter 8

Conclusion

Résumé

Dans la thèse, nous avons présenté un nouvel ensemble d'outils pour analyser, comprendre et transformer les programmes écrits dans des environnements propriétaires tels que MATLAB. Notre approche est basée sur des profils d'événements de performance qui enregistrent la façon dont un programme s'exécute directement sur le processeur sous-jacent. Cette connaissance nous permet de contourner l'environnement propriétaire de MATLAB. De plus, nous avons intégré le comportement observé de l'exécution des programmes dans MATLAB dans un modèle d'exécution basé sur un arbre. En nous basant uniquement sur ce modèle, nous avons proposé plusieurs transformations de code qui augmentent les performances des programmes. Ces transformations comprennent la vectorisation de boucle guidée par le profil, la restructuration des copies de tableaux et l'amélioration de l'indexation des tableaux. Enfin, nous avons rassemblé ces transformations sous un seul outil, un compilateur HU!M qui permet la restructuration automatique du code MATLAB.

Notre méthodologie d'analyse est également applicable à d'autres langages populaires tels que Python, Octave ou Julia. Les travaux futurs comprennent des améliorations du compilateur HU!M et des travaux supplémentaires sur le modèle d'exécution pour le transformer en un modèle de performance à part entière.

Introduction

MATLAB is a highly popular, easy to use, language and environment for scientific computing, commonly used for image processing, radar design, machine learning, and many others disciplines. Unfortunately, the inherent easiness of writing programs in MATLAB contrasts with its attainable performance, especially when compared to statically compiled programs in C and Fortran languages. Therefore, in order to combine the benefits of developing compute-intensive applications written in MATLAB with achieving a reasonable performance, many researchers have worked on techniques for increasing performance of MATLAB programs.

The most common approach to improve MATLAB programs is compilation to compiled languages, such as C or Fortran, which have already

performant optimising compilers. However, this approach requires, each time, a compilation of the MATLAB program before running it. The compilation takes time and requires switching to a different toolchain. Therefore, in our work, we have focused on improving MATLAB programs through code transformations which are applied only once and can be performed without leaving the MATLAB environment.

8.1 Summary

The thesis consists of three interleaved parts: (1) the description of our analysis “tool”, *performance event profiles* in Chapter 3; (2) application of the “tool” to build an execution model of MATLAB expressions in Chapter 4; and (3) proposition of new code transformations for MATLAB programs in Chapter 5, based on gained knowledge and the execution model.

Contributions

The utilisation of *performance event profiles* (1) in Chapter 3 was necessary because MATLAB is a black-box, therefore, we possess no knowledge about how MATLAB analyses, schedules, and executes instructions. Instead, we have focused on a processor and its hardware performance counters which record the program execution directly on the processor (see Figure 3.1). Therefore, *performance profiles* describe how the program performance changes as the program execution progresses. Moreover, we have introduced a notion of *execution region* describing a segment of the profile with interesting performance properties (in Section 3.4). Thus, allowing us to clearly see what are the smaller components of the program execution, what is their order, and how they behave.

As it turned out later, the ability to see time-varying behaviours during the execution of MATLAB programs is the key component to understand how MATLAB really executes programs. In Chapter 4, using performance profiles, we have analysed several programs only to observe that the JIT compiler combines some instructions together, while leaving others apart. This has led to a concept of *instruction block*, a collection of instructions scheduled for combined execution by the JIT compiler (in Section 4.2). Furthermore, we have encoded the rules of merging instructions into blocks and how these blocks form an *instruction tree* (in Algorithm 2). The instruction tree from Section 4.5 in its minimal form (with the smallest possible number of instruction blocks) constitutes our execution model which correctly predicts the number, order, and kind of instruction blocks for an input MATLAB expression (see an example prediction in Figure 4.9).

With the gained knowledge about program execution in MATLAB and with our execution model, we have proposed in Chapter 5 several code transformations which restructure programs, e.g. *repacking of array slicing*,

range simplification or replace language constructs, e.g. *dynamic array slicing*, *fast array slicing*. Moreover, the presented execution model is a basis for further code transformation described in the future work. Furthermore, in the same chapter, we have explored the potential of applying profiling to loop vectorisation.

Implementation

Apart from the conceptual contributions, during the work on the thesis, we have implemented three important tools: **HU!M**, our source-to-source compiler which implements the execution model and the presented code transformations; **mPAPI**¹ which allows to access hardware performance counters from MATLAB; and **Menchi** which generates experiments from a single specification file.

Results

Presented code transformations show encouraging results. For example, the *repacking of array slices* is able to speedup programs up to 80% (see Figure 5.2). However, before predicting when we should apply the *repacking*, our model needs further improvements. Furthermore, the profile-guided approach to loop vectorisation presented in Section 5.4, combined with *range simplification*, was able to improve three benchmarks: **crni** by 25.7%, **pagerank** by 33.3%, and **backprop** by 45.8%. In comparison, for all three benchmarks, the approach by Chen et al. [1] resulted in performance slowdowns.

Perspective

The code transformations and the execution model, presented in this thesis, are specific to MATLAB. However, the performance event profiles (PEP) could be used in other runtime environments, because the profiles are concerned with how the program executes on a processor, and not necessary about what is inside the runtime. Moreover, it is possible that the methodology of discovering which built-in functions are JIT-compiled presented in Section 4.4 is applicable to other languages too.

To our best knowledge, the work presented in this thesis is the first attempt to build an execution model for programs running in the MATLAB environment.

8.2 Future work

Our research work has created a few areas of improvements in performance event profiles, execution model, on the presented code transformations, and

¹<https://github.com/quepas/mPAPI>

in the implementation of our compiler HU!M.

Performance event profiles (PEP). In our work, we have used only a handful of available native performance events on our processors (there are 171 native events on machines M1 and M2; see Table A.1). Other events include information about various segments of the processor pipeline or even how particular μ -ops were distributed among execution ports. All these events and their possible combinations create an untapped potential for designing new insightful performance event profiles (PEP) that deserve to be explored.

Furthermore, because the idea behind *performance event profiles* is to skip the execution environment and measure the execution directly on a process, we would like to test our methodology for two other environments: Julia [136] and PyPy JIT compiler for Python [138].

Execution model. There are two possible major improvements to the proposed execution model. The first one is the special consideration of instruction blocks which mix scalar and vector operations. In such blocks, the JIT compiler is forced to issue a scalar code even for vector instructions, because it cannot issue the vector code for the scalar instructions. By splitting an expression in such situations, we can break the mix of scalar and vector operations, thus, letting the vector operations to execute at full speed.

The second extension is the consideration of multi-threading. One of our previous works has shown that multi-threaded built-in functions divide the computation between many threads in several predictable patterns, e.g. (1) equal split of the computation between threads, or (2) the first thread takes 50 % of the computation, and the rest of the computation splits equally between other threads. We would like to encode those patterns inside our model to predict how the JIT compiler schedules execution of blocks on many threads.

Code transformations. As seen in Section 5.2, repacking of array slices is not always beneficial. Therefore, we plan to adjust our execution model from Chapter 4 to precisely predict the benefits of the repacking transformation. Moreover, we would like to further develop our ongoing work on code transformations which reorder, fuse, and split expressions in order to change the number of instruction blocks created by the MATLAB JIT compiler, or to create blocks with specific properties (e.g. which contain only vector instructions). We think, these transformations are crucial for generating optimal MATLAB code.

HU!M compiler. Apart from further work on the general capabilities of our compiler such as type analysis, we would like to implement in HU!M three loop transformations: loop tiling [27], scalar replacement [85], and loop

unrolling [160]. Our previous works have shown promising results for these three transformations.

Appendix A

Experiment methodology

In the thesis, we use a consistent experiment methodology which contains two parts: (1) preparation of the environment; and (2) collecting measurements. The part (1) is concerned with minimising the measurement error and non-deterministic events [145]. In the part (2), we are interested in how to collect measurement in order to obtain representative results [124].

A.1 Preparation of the environment.

In this step, we follow `krun` tool [162] from the work of analysing warm-up states of virtual machines by Barrett et al. [36]. `krun` is an extreme benchmark runner which prepares the machine and the operating system for performing precise measurements of computer programs. The tool is capable of restarting the machine before each run and running benchmarks only when the CPU temperature is below a given threshold. In order to limit the time required for running all tests, we only reuse some of the less extreme techniques:

- Restarting machines before running batch of experiments.
- Running experiment scripts only from command line without X11 (GUI) turned on.
- Turning off active frequency scaling (Intel Turbo Boost), by setting up `performance` governor which keeps the frequency at the base level (`sudo cpufreq-set -g performance`).
- Synchronising buffered data on disk (`sudo sync`).
- Sleeping for 5 seconds before starting the benchmark (to stabilise the system after recent command; `sleep 5`).
- Running benchmark scripts with the highest priority (`sudo nice -n -20`).

A.2 Collecting measurements

Each of our experiments consists of at least 30 repeated measurements of the same phenomena. Even when we report a single performance profile, the profile comes from the last repetition (like in Figures 3.3, 3.4 and 4.2). In the case of measurements in counting mode, we either report each measurement and show its data distribution using: box-plot (in Figure 3.5) or violin plot (in Figure 3.7); or we compute confidence intervals and show them along with the mean value (in Figure 5.2). The confidence intervals help to find only relevant data without outliers [42]. We compute them using bootstrap method (3000 trials), because we do not feel confident to assume the normal distribution for the measurements of processor cycles and real time which usually are skewed towards the maximal value (if a non-deterministic event occurs, it will add to the execution time or processor cycles) [163].

A.3 Machine specification

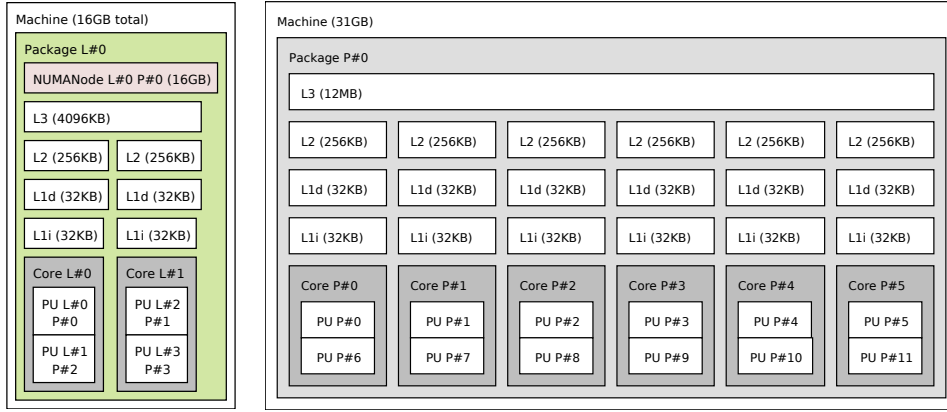
All experiments and plots in the thesis were prepared on two machines with Intel processors. Table A.1 outlines their detailed specification. Figure A.1 presents topology of Intel processors in M1 and M2 machines (processor cores and the hierarchy of cache memories).

Table A.1: Specification of two machines M1 and M2 used in our work.

Property	Machine	
	M1	M2
Vendor	HP Inc.	Dell Inc.
Model	EliteBook 830G	Precision Tower 3430
Class	Notebook	Desktop computer
CPU model	Intel® Core™ i7-6600U	Intel® Core™ i7-8700
Microarchitecture	Skylake	Coffee Lake
Frequency (min/base/max)	0.4 GHz/2.6 GHz/3.4 GHz	0.8 GHz/3.2 GHz/4.6 GHz
Cores (physical/logical)	2/4	6/12
Data Caches (L1/L2/L3)	32 kB/256 kB [†] /4096 kB* [†]	32 kB/256 kB [†] /12 288 kB* [†]
RAM (size/frequency/type)	16 GB/2133 MHz/DDR4	32 GB/2666 MHz/DDR4
OS	Ubuntu 16.04.3 LTS	Ubuntu 18.04.3 LTS
Kernel	4.4.0-159-generic	5.0.0-29-generic
PAPI	5.7.1.0	5.7.1.0
Perf. events (native/preset)	171/59	171/59

* Shared between cores.

[†] Unified cache (data and instructions).



(a) Machine M1

(b) Machine M2

Figure A.1: Hierarchical topology of two machines M1 and M2 respectively.

Appendix B

Measuring performance events in MATLAB with mPAPI

`mPAPI`¹ is our open source tool for measuring hardware performance counters directly in MATLAB code (it also works in Octave). The tool is built using PAPI library [43, 95], a popular and powerful interface for accessing performance counters. PAPI has a C interface, therefore, we use MEX files to access its low-level routines from MATLAB.

The motivation behind creation of `mPAPI` comes from a lack of tools for easy access to hardware performance counters. Popular tools like `perf` [164] and Intel VTune [99] give us two choices for measuring performance counters: (1) measuring MATLAB program from the beginning including the warm-up of the MATLAB environment; or (2) attaching to already running process and measuring counters only for it. Option (1) is easy to automate, but it is inaccurate because we always measure the warm-up of the MATLAB environment. Second option (2) is hard to automate, because we need to attach to the running MATLAB environment in a particular moment, after the warm-up. Therefore, registering and measuring performance counters directly from the MATLAB code is the easiest and most reliable approach.

PAPI has an interface for MATLAB, however, it allows only to measure performance counters in the counting mode. `mPAPI` extends the functionality with three additional features:

1. Multiplexing — measuring many performance events at once [91].
2. Per-thread measurements — collecting performance counters from many threads (processes).
3. Performance profiles — measuring performance events in the sampling mode (we use here Event-Based Sampling to be precise) [91, 132].

¹<https://github.com/quepas/mPAPI>

Each function from the mPAPI interface is persistent, which means that it cannot be removed from the MATLAB workspace, once loaded. This is a requirement for our experiment methodology, which clears all loaded functions (especially JIT-compiled functions) on the change of test data. This is due to the fact that when testing MATLAB code with a small data, the code is JIT-compiled and saved for later execution, making the initial runs very slow in comparison to even later one with bigger data sets (because by now, the test code is JIT-compiled).

B.1 mPAPI interface.

This section describes a list of available functions from mPAPI tool.

B.1.1 Enumerating available performance events

This functionality mimics two tools from PAPI toolchain: (1) `papi_avail -d` for listing *preset events* and (2) `papi_native_avail` for enumerating *native events*. Processors use native events which represent very specific events available on a given microarchitecture. However, in the PAPI ecosystem, there are also preset events which try to generalise common performance events and concepts. For example, two microarchitectures μ_1 and μ_2 might have two different native events X and Y for measuring the same number of load instructions. Therefore, instead of measuring two specific native event, we can always measure single preset event `PAPI_LD_INS` which maps to X on μ_1 and to Y on μ_2 microarchitecture.

```

1 % Returns an array of names for native event
2 mPAPI_enumNativeEvents()
3 % Returns an array of names for preset event
4 mPAPI_enumPresetEvents()
```

On our test machines M1 and M2 (see Table A.1), `mPAPI_enumNativeEvents` returns 171 events and `mPAPI_enumPresetEvents` returns 59 events.

B.1.2 Measuring performance events in counting mode

Measuring performance events in mPAPI (for counting and sampling modes) always contains the same three step workflow: (1) register performance events to measure; (2) start the measurement; and (3) finish the measurement and collect results. Step (1) have three functions dedicated to registering an event set indicating what should be measured:

```

1 % Register events for the current thread
2 event_set = mPAPI_register(events_list)
3 % Register events for the current thread with multiplexing
4 event_set = mPAPI_register(events_list, true)
```

```

5 % Register events for the process pid
6 event_set = mPAPI_register(events_list, pid)

```

The `events_list` can contain a mix of native and preset events for the measurement. In the case of per-process measurement, we need to call `mPAPI_register` once for each `pid` and obtain several event sets. An event set contains the list of performance events, but also information about the measured process and if the measurements are gathered using multiplexing.

```

1 % Start measurement
2 mPAPI_tic(event_set)
3 mPAPI_tic([event_set_1, event_set_2])
4 % Stop measurement and collect results
5 results = mPAPI_toc(event_set)
6 results = mPAPI_toc([event_set_1, event_set_2])

```

The `mPAPI_tic` function can take one or more event sets and starts the measurement. We stop the measurement and collect results using `mPAPI_toc` which takes the same one or more event sets representing ongoing measurements.

B.1.3 Measuring performance events in sampling mode

The exact procedure of creating performance profiles and using `mPAPI` to measure performance traces is detailed in Section 3.3.3. Measuring performance events in sampling mode mainly differs from counting mode with the fact that samples are stored in an external *profile file*, instead of being returned inside the MATLAB workspace. This is due to the fact that the single performance trace can contain up thousands of measurements, thus, it requires a post-processing in external tools.

As before, the workflow of measuring performance counters stay the same: registering of performance events followed by start and stop of the measurement procedure. However, now the functions have more arguments. The function for registering performance trace takes 4 arguments: the sampling event, sampling threshold, list of performance events to measure, and the path to profile file. Moreover, the measurement start function takes one additional argument for the name of the current trace. Each of these arguments is described and explained in Section 3.3.3.

```

1 % Register performance trace
2 event_set = mPAPI_trace_register(sampling_event, sampling_threshold,
   events_list, trace_file)
3 % Start the measurement with a given name
4 mPAPI_tic(event_set, trace_name)
5 % Stop the measurement and save the current trace in trace-file
6 mPAPI_toc(event_set)

```


Appendix C

Automatic experiment generation with Menchi

Menchi is an experiment generator for automatic benchmarking of MATLAB/Octave programs. The goal of **Menchi** is to facilitate repetitive tasks such as generating scripts, preparing a machine for tests, and running benchmarks with changing experiment parameters. Moreover, the tool makes sure that all experiments share the same experimental conditions to ensure benchmarking repetitiveness.

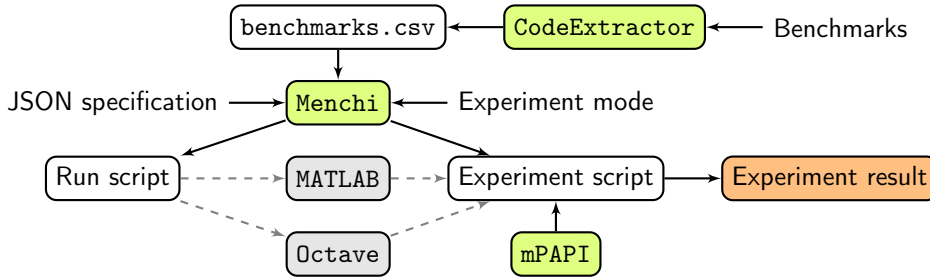


Figure C.1: Workflow of code benchmarking with **Menchi**. Three olive nodes highlight our tools: **CodeExtractor**, **Menchi** and **mPAPI**.

Figure C.1 presents steps and tools required to prepare experiments with **Menchi**. We start with preparing and extracting source code of benchmarks to the CSV file format. An auxiliary tool called **CodeExtractor** takes a set of benchmark codes, analyses them, and packs them into one CSV. Next, we define the experiment specification in one JSON file roughly divided into three sections: (1) environment, (2) experiment, and (3) measurement. The environment section (1) contains paths to MATLAB/Octave environments, **mPAPI** tool and paths to any external C MEX libraries used in the benchmark codes, e.g. **MatlUp** with **mu_fast_slice** functions. The experiment section (2) defines properties such as: experiment name, path to a result files, MAT-

LAB/Octave versions, benchmark codes, or the size of input data. The measurement section (3) declares what is measured (execution time and/or performance events), and how many times to repeat measurements.

Finally, we pass to **Menchi** two parameters with a path to the JSON specification and the experiment mode (e.g. counting or sampling) as follows: `menchi spec.json experiment_mode`. The path to benchmark codes is inside the specification. The result of running **Menchi** is a collection of generated runnable scripts.

In the subsequent sections, we describe how to prepare benchmark codes and experiment specification to generate run scripts and execute benchmarks. Moreover, we list 8 available experiment modes with their description.

C.1 Benchmark preparation

Menchi loads benchmarks from CSV files prepared using **CodeExtractor** tool. The tool takes as an input files with benchmark codes structured as separate MATLAB functions. Listing C.1 presents an example benchmark from the Livermore suite¹. Each benchmark code consists of several code regions marked with the `pragma` directive. The only required region is `init` which sets values to variables used in the benchmark code. Usually, the `init` section performs random value generation of vectors with `LEN_1D` elements and matrices of size `LEN_2D × LEN_2D`. All subsequent code regions contain different versions of the same benchmark, e.g. `loop`, `vec`, `vec_01`, `unrolled_2`, `tiled`.

Listing C.1: Structure of the benchmark source code used by the **CodeExtractor** and **Menchi**. The example code contains three versions (`loop`, `unrolled_2`, `vec`) of the inner product kernel from the Livermore suite.

```

1 function kernel_3_inner_prod(LEN_1D)
2     %! pragma init
3     q = 0.0;
4     x = randn(1, LEN_1D);
5     z = randn(1, LEN_1D);
6     %! pragma loop
7     for k = 1:LEN_1D
8         q = q + z(k) .* x(k);
9     end
10    %! pragma unrolled_2
11    for k = 1:2:(LEN_1D-1)
12        q = q + z(k) .* x(k);
13        q = q + z(k+1) .* x(k+1);
14    end
15    %! pragma vec
16    q = sum(z(1:LEN_1D) .* x(1:LEN_1D));

```

¹<https://www.netlib.org/benchmark/livermore>

17 **end**

The **CodeExtractor** tool takes benchmark codes and extracts them to a single CSV file containing 9 columns listed in Table C.1 with their descriptions. Apart from standard information about benchmark name, versions, and their source code, **CodeExtractor** extracts information about the number of vector (**num_vectors**) and matrix (**num_matrices**) variables defined in the **init** region. The information is later used by the **Menchi** to generate input variables which occupy a specific amount of memory, e.g. all L1 cache or half of the L2 cache. Moreover, the tool keeps lists of variables defined in the **init** region and in source codes of every benchmark version. Later, **Menchi** uses the lists to empirically validate if two or more benchmark versions generate the same result, which is a useful feature while benchmarking new code transformations.

Table C.1: Description of the CSV benchmark specification generated by the **CodeExtractor**.

Column	Description
benchmark	Benchmark name extracted from the function name in which the benchmark is stored
type	Benchmark version from the pragma directive (except for the init region)
parameter	The name of the data size parameter (LEN_1D or LEN_2D)
num_vectors	Number of vector variables defined in the init region
num_matrices	Number of matrix variables defined in the init region
init	Source code of the init region which sets values of input variables, common for every benchmark version
code	Source code of the benchmark version region
init_vars	List of variables defined in the init region
code_vars	List of variables defined in the benchmark version region

C.2 Experiment specification

In **Menchi**, a single specification file defines all properties of an experiment except for the experiment mode which is an additional parameter to the **Mechi** tool. The specification file is in the JSON format with predefined structure and field names. The file contains three groups of parameters describing: environment (**env**), experiment (**experiment**), and measurements (**measurements**). Appendix C.2 depicts example specification of an experiment for the measurement of vectorised codes from the **STREAM** benchmark.

1 {


```

2  "env": {
3    "MATLAB": {
4      "R2018b": "/usr/local/MATLAB/R2018b/bin/matlab",
5      "R2015b": "/usr/local/MATLAB/R2015b/bin/matlab"}},
6    "mPAPI": {
7      "R2018b": "../../Tools/mPAPI/r2018b",
8      "R2015b": "../../Tools/mPAPI/r2015b"}},
9    "notification": {
10     "enabled": 1,
11     "credentials": "/home/quepas/pushover_credentials.json"}
12  },
13  "experiment": {
14    "name": "stream_vectorised_performance",
15    "directory": {
16      "result": "results/",
17      "temporary": "temp/"},
18    "execution": {
19      "hyper-threading": true,
20      "process_iter": 1,
21      "threads": [1, 2, 4],
22      "matlab": ["R2015b", "R2018b"]},
23    "N": {
24      "enumerate": "L1, L2, 2*L2, 4*L2"},
25    "benchmarks": {
26      "input_codes": [
27        "../../Benchmarks/_extracted/STREAM.csv"],
28      "version": ["vec"],
29      "include": [],
30      "exclude": []},
31    "measurement": {
32      "repetitions": 30,
33      "test_time": true,
34      "performance_events": [["CPU_CLK_UNHALTED:THREAD_P",
35                             "INST_RETIRED:ANY_P"]]}
36  }
37 }

```

Menchi prepares the experiment files accordingly to our experiment methodology detailed in Appendix A. For example, the tool sets up the constant frequency of the processor (performance governor), clears I/O buffers, sets the highest processes priority, among others. Moreover, Menchi can send push notification using Pushover service to notify when the test have finished.

C.3 Experiment modes

Menchi generates 8 types of experiments depicted in Table C.2, mainly used for the measurement of execution time and performance events. Moreover, the tool collects data from code instrumentation and performs empirical validation of results generated by different versions of the same benchmark.

Table C.2: Experiment modes in `Menchi` tool.

Experiment code	Code placement	<code>Menchi</code> parameter
Measurement	Script	<code>measure</code>
Measurement	Function	<code>measure_fun</code>
Per-thread measurement	Script	<code>threads</code>
Per-thread measurement	Function	<code>threads_fun</code>
Warmup	Script	<code>warmup</code>
Performance profile	Function	<code>phase</code>
Instrumentation	Script	<code>instrument</code>
Empirical validation	Script	<code>validate</code>

Measurement mode. In this mode, we can measure the execution time using `tic` and `toc` built-in functions which have a resolution of 1×10^{-6} seconds. Moreover, we can measure performance events using `mPAPI` tool which accesses native performance events and PAPI present events in the counting mode.

The results of the *measurement mode* are stored in a long (narrow) data format and contain several execution properties listed in the header in Listing C.2. Apart from obvious entries: `matlab`, `threads`, `benchmark`, `metrics`, `N`, and `value`, the properties include `process`: for counting how many times the MATLAB environment was restarted; `in_process` for counting how many times a measurement was taken; and `version` which indicates the code version of a benchmark.

Listing C.2: Example of benchmark results in a long format.

```

1 matlab,threads,process,benchmark,version,metrics,N,in_process,value
2 R2018b,1,1,add,loop,INST_RETIRED:ANY_P,1000000,1,126562546
3 R2018b,1,1,add,loop,MEM_INST_RETIRED:ALL_LOADS,1000000,1,36292657
4 R2018b,1,1,add,loop,INST_RETIRED:ANY_P,1000000,2,21222667
5 R2018b,1,1,add,loop,MEM_INST_RETIRED:ALL_LOADS,1000000,2,3066188
6 ...

```

Per-thread measurement mode. The mode is similar to the *measurement mode*, but it performs measurements per-thread basis. Therefore, the result file in Listing C.3 contains an additional execution property, the `thread` number with values from the range of `[1, threads]`.

Listing C.3: Example of benchmark results in a long format with per-thread measurements.

```

1 matlab,thread,threads,process,benchmark,version,metrics,N,in_process,
  value
2 R2018b,1,2,1,add,loop,INST_RETIRED:ANY_P,1000000,1,118316140

```

```

3 R2018b,2,2,1,add,loop,INST_RETIRED:ANY_P,1000000,1,0
4 R2018b,1,2,1,add,loop,INST_RETIRED:ANY_P,1000000,2,21224014
5 R2018b,2,2,1,add,loop,INST_RETIRED:ANY_P,1000000,2,0
6 ...

```

Function or script sub-modes. Both modes, *measurement* and *per-thread measurement*, have two sub-modes where measured benchmarks are embedded in either, a script or a separate function. The placement of benchmarks affects their execution and performance, because the sub-modes use MATLAB workspace differently. In the script, every time the benchmark stores a variable, the global workspace is updated as well which generates additional size and type checks among others. Potentially, the placement of benchmarks affects capabilities of the JIT compiler too, because every assignment in the script modifies the workspace, where in the function, only assignments which return a result.

From our experience, measuring benchmarks embedded in the function is more precise. However, depending on the context, measuring in the script or in the function is a more appropriate approach.

Warmup measurement mode. The primary use of *warmup measurement* mode is to examine how program execution changes after the program runs multiple times. Modern Just-In-Time compilers can monitor how the same program executes and recompile the program if profitable. This methodology is an example of Profile-Guided Optimisation (PGO) found in e.g. HotSpot JVM [165, 166]. The goal of the *warmup measurement* mode is to accommodate a huge number of measurements in a wide format depicted in Listing C.4 and save storage memory in the process.

Listing C.4: Example of benchmark results in a wide format for warmup measurements.

```

1 matlab,threads,process,benchmark,version,metrics,N,1,2,3,4,5, ...
2 R2018b,add,loop,1,1000000,INST_RETIRED:ANY_P
  ,170517051,48110361,48098964,48097041,49076418, ...
3 R2018b,add,vec,1,1000000,INST_RETIRED:ANY_P
  ,26513485,25383927,25383917,25383815,27349558, ...
4 R2018b,add,vec_01,1,1000000,INST_RETIRED:ANY_P
  ,5993393,5595819,5595819,5595820,6526795, ...

```

Performance profile mode. In the *performance profile* mode, Menchi collects performance traces using the mPAPI in the sampling mode. Example of a performance trace is depicted in Listing 3.2 in Chapter 3. However, traces are hard to process and analyse because they store raw measurements. Therefore, with the `trace2csv()` function from mPAPI, traces are converted to a single CSV file in a long format.

The result CSV file has the same header as in the *measurement* mode in Listing C.2, but with two additional columns: `trace_id` and `time`, because each trace has its unique identifier (from 1 to N) and the observation time from the sampling. Due to limitations of hardware performance counters, there is no per-thread measurement in the *performance profile* mode.

Code instrumentation mode. Apart from classical measurements, **Menchi** also collects data from the code instrumentation. We have developed a tool which replaces language constructs with wrapped functions to count the occurrence of `read`, `write`, `flops` instructions in the code. The result file is in a long format (as in Listing C.2) with the same header as in the *measurement* mode, but with only three metrics: `read`, `write`, `flops`.

In the current version, the set of collected metrics by **Menchi** is fixed to the three above-mentioned metrics. However, it is possible to add new metrics in the instrumentation tool and **Menchi**.

Empirical validation mode. In principle, each code version of a benchmark returns the same result (see Listing C.1). However, in the process of preparing automated (or manual) code transformations, an implementation bug might occur resulting in the invalid transformations. The goal of the *empirical validation* mode is to check if various code versions perform the same computation.

In the mode, for each measurement repetition, **Menchi** generates and saves input data for the run. Then, the tool executes each code version with the same input data and stores their results. Finally, the obtained results are compared and in the case of a result mismatch, the tool displays an error. **Menchi** performs the numerical comparison with a tolerance defined by the parameter `tolerance` in the JSON specification of the experiment.

Appendix D

Accompanying materials

This section contains several accompanying materials such as: code listings on Figure D.1 used in the profile-guided loop vectorisation in Section 5.4; extended results on Figure D.2 of the cost analysis of array slicing in Section 3.5; and finally, performance event profiles (PEP) for code examples from Table 4.4 on Figures D.3 to D.8.

```
1 function crni2(LEN_2D)
2     %! pragma init
3     Vb = zeros(1, LEN_2D);
4     U = randn(LEN_2D, LEN_2D);
5     s2 = randn(1);
6
7     %! pragma loop
8     for j1 = 2:(LEN_2D+1)
9         for i1=2:(LEN_2D-1)
10             Vb(i1)=U(i1-1, j1-1)+U(i1+1, j1-1)+s2*U(i1, j1-1);
11         end
12     end
13
14     %! pragma mc2mc
15     for j1 = 2:(LEN_2D+1)
16         i1 = colon(2,minus(LEN_2D,1));
17         if length(Vb)==length(i1)
18             Vb=plus(plus(U(minus(i1,1),minus(j1,1)),U(plus(i1,1),minus(j1,1))),times(minus(
19                 rdivide(2,s2),2),U(i1,minus(j1,1))));
20         else
21             Vb(i1)=plus(plus(U(minus(i1,1),minus(j1,1)),U(plus(i1,1),minus(j1,1))),times(minus(
22                 rdivide(2,s2),2),U(i1,minus(j1,1))));
23         end
24     end
25
26     %! pragma mc2mc_opt
27     for j1 = 2:(LEN_2D+1)
28         Vb(2:(LEN_2D-1))=U(1:(LEN_2D-2), j1-1)+U(3:LEN_2D, j1-1)+s2*U(2:(LEN_2D-1), j1-1);
29     end
30 end
```

Figure D.1: Loop `crni2` from the work by Chen et al. [1] and its vectorised versions, without and with our code transformations.

```

1  function nw2(LEN_2D)
2      %! pragma init
3      input_itemsets = randn(LEN_2D, LEN_2D);
4      jj = randi([1, LEN_2D], 1, 1);
5      penalty = randn(1);
6
7      %! pragma loop
8      for ii = 2:LEN_2D
9          input_itemsets(ii, jj) = -(ii - 1) * penalty;
10     end
11
12     %! pragma mc2mc
13     ii = colon(2, LEN_2D);
14     input_itemsets(ii, jj) = times(uminus(minus(ii, 1)), penalty);
15
16     %! pragma mc2mc_opt
17     input_itemsets(2:LEN_2D, jj) = -(1:(LEN_2D - 1)) .* penalty;
18 end

```

```

1  function nw3(LEN_2D)
2      %! pragma init
3      input_itemsets = randn(LEN_2D, LEN_2D);
4      ii = randi([1, LEN_2D], 1, 1);
5      penalty = randn(1);
6
7      %! pragma loop
8      for jj = 2:LEN_2D
9          input_itemsets(ii, jj) = -(jj - 1) * penalty;
10     end
11
12     %! pragma mc2mc
13     jj = colon(2, LEN_2D);
14     input_itemsets(ii, jj) = times(uminus(minus(jj, 1)), penalty);
15
16     %! pragma mc2mc_opt
17     input_itemsets(ii, 2:LEN_2D) = -(1:(LEN_2D-1)) * penalty;
18 end

```

Figure D.1: Two loops `nw2` and `nw3` from the work by Chen et al. [1] and their vectorised versions, without and with our code transformations.

```

1 function fft1(LEN_2D)
2     %! pragma init
3     rtnR = zeros(LEN_2D, LEN_2D);
4     rtnI = zeros(LEN_2D, LEN_2D);
5     resR = randn(1, LEN_2D);
6     resI = randn(1, LEN_2D);
7
8     %! pragma loop
9     for ii = 1:LEN_2D
10         for k=1:LEN_2D
11             rtnR(ii,k) = resR(k);
12             rtnI(ii,k) = resI(k);
13         end
14     end
15
16     %! pragma mc2mc
17     for ii = 1:LEN_2D
18         k = colon(1,LEN_2D);
19         rtnI(ii, k) = resI(k);
20         rtnR(ii, k) = resR(k);
21     end
22
23     %! pragma mc2mc_opt
24     for ii = 1:LEN_2D
25         rtnR(ii, :) = resR(1:LEN_2D);
26         rtnI(ii, :) = resI(1:LEN_2D);
27     end
28 end

```

Figure D.1: Loop `fft1` from the work by Chen et al. [1] and its vectorised versions, without and with our code transformations.

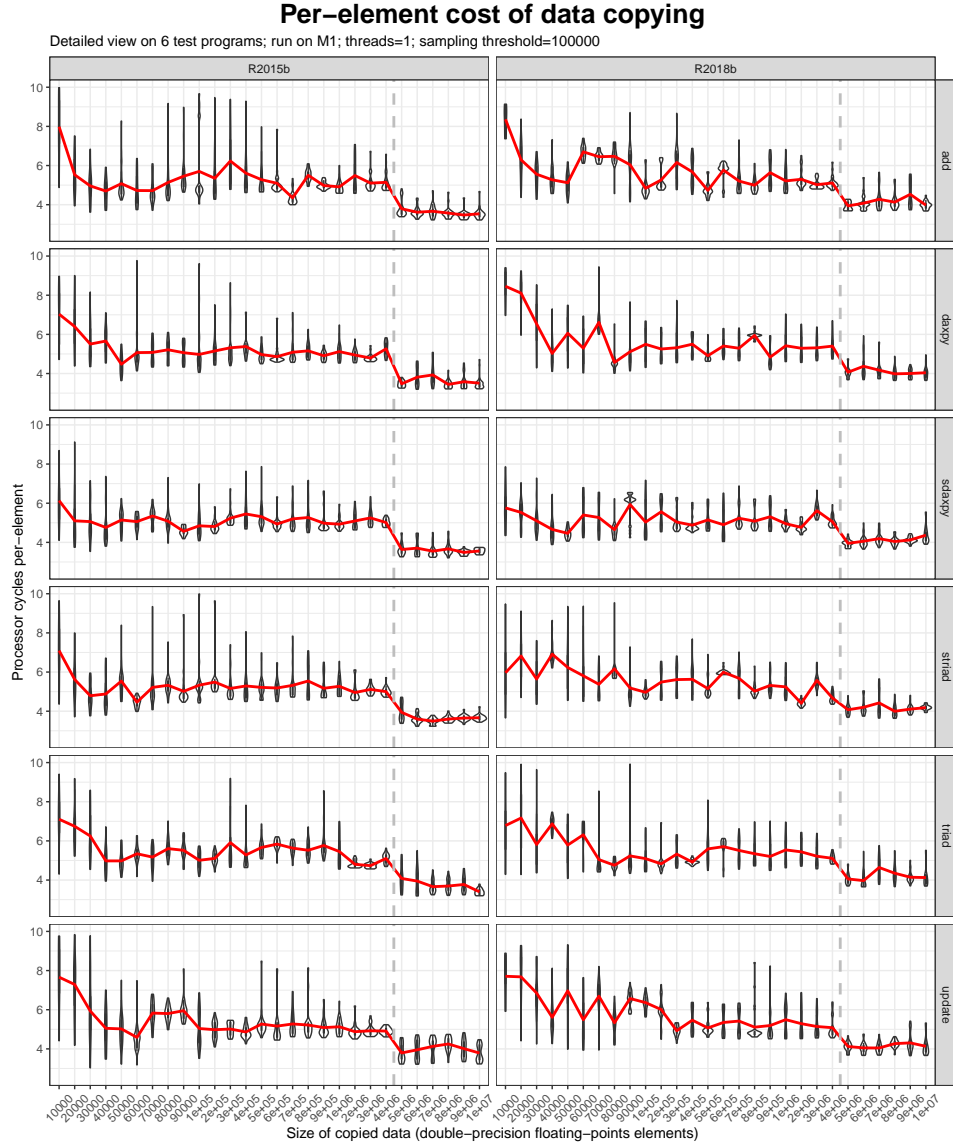


Figure D.2: The per-element cost of performing data copy during *array slicing*. The results show the copy of huge volume of data (more than 4000 000 elements) is performed more efficient in terms of processor cycles for all 6 programs.

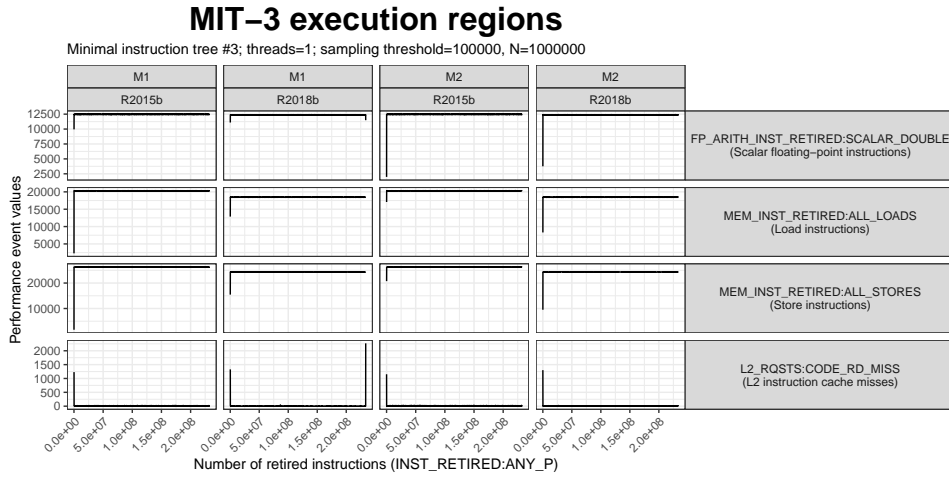


Figure D.3: Expression $\text{floor}(A) + \sin(\text{fix}(B * C))$ is perfectly combinable and requires only one instruction block to execute.

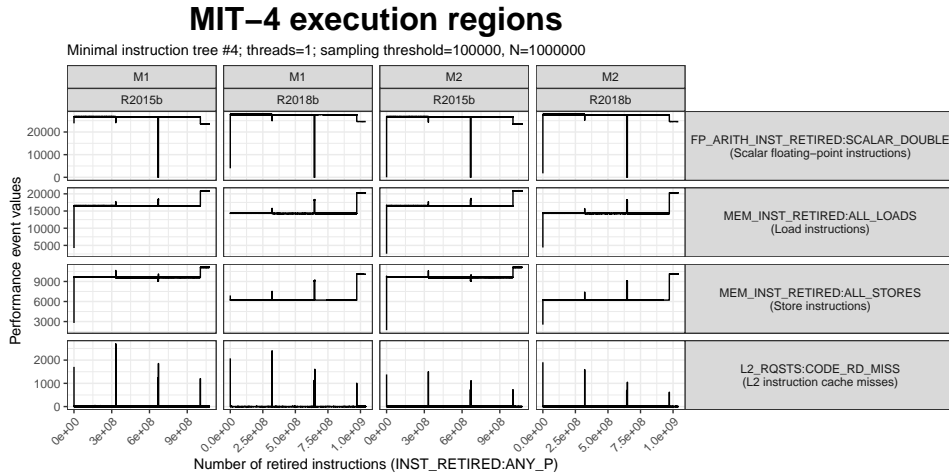


Figure D.4: Expression $\exp((A.^D + B.^E) ./ (C.^F))$ contains compute-intensive power operators.

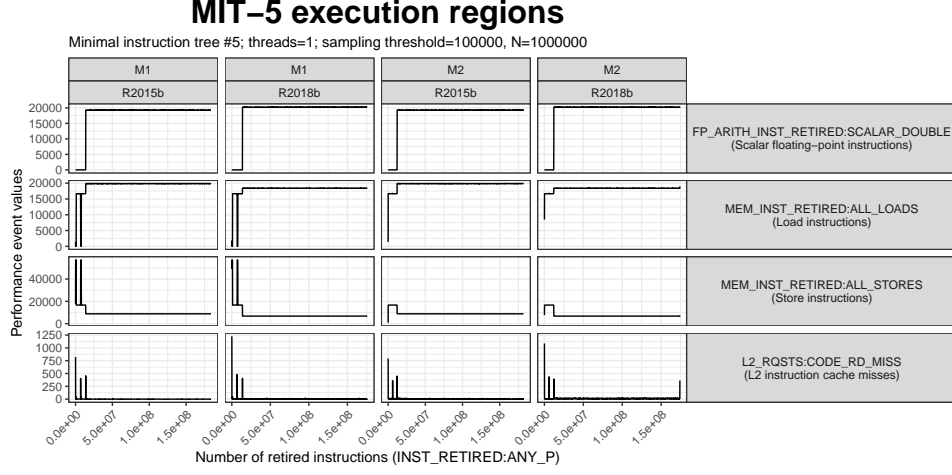


Figure D.5: Expression $A(1:N) .* \text{atan2}(B(1:N), C)$ contains atan2 function which combined with element-wise multiplication $.*$ creates an instruction block performing scalar arithmetic operations only.

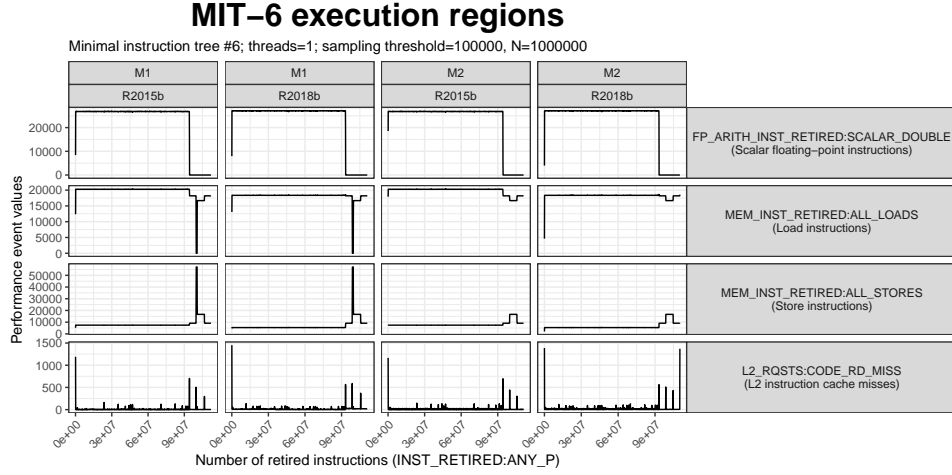


Figure D.6: Expression $\log(A) + B + C(1:N)$ performs compute-intensive scalar \log function followed by the addition of B with the array slice $C(1:N)$.

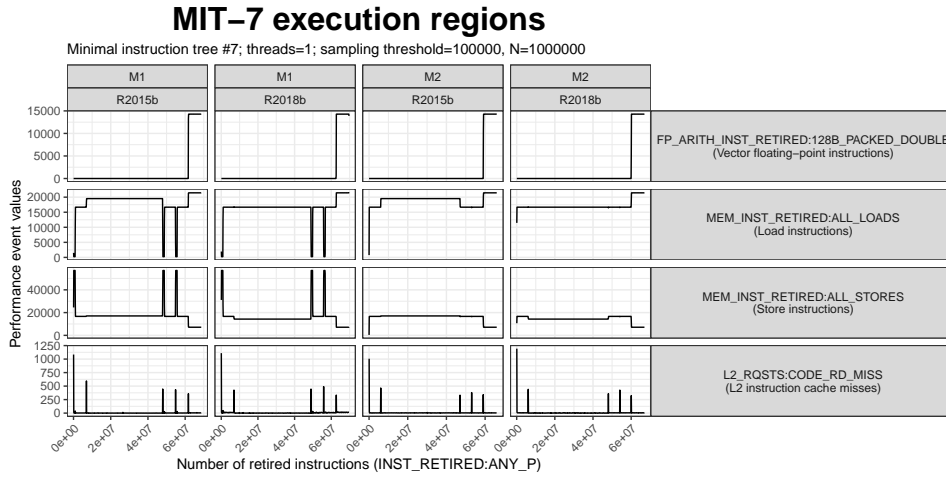


Figure D.7: Expression $\text{fix}(A(1:N)) + (B(1:N) .* C(1:N))$, when executed, it mixes addition and element-wise multiplication into one instruction block performing vector arithmetic operations.

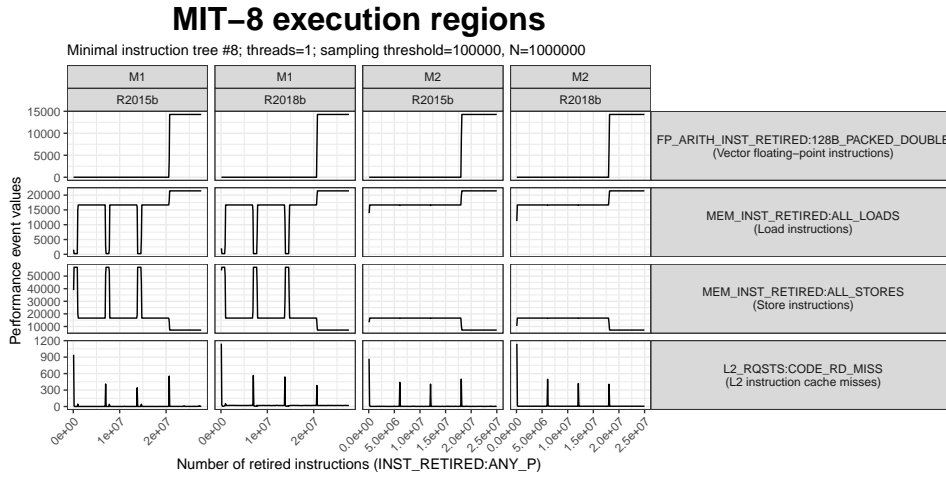


Figure D.8: Expression $A(1:N) + (B(1:N) + C(1:N))$ evaluates chain of additions in the right-to-left order; thus, allowing plus operators to coexist in a single instruction block.

Bibliography

- [1] Hanfeng Chen, Alexander Krolik, Erick Lavoie, and Laurie Hendren. Automatic Vectorization for MATLAB. volume 10136 LNCS of *Lecture Notes in Computer Science*, pages 171–187. 2017.
- [2] Kenneth E. Iverson. *A programming language*. Wiley, 1962.
- [3] Julia Language: Microbenchmarks. <https://github.com/JuliaLang/Microbenchmarks>. Accessed on: 2019-09-28.
- [4] Efstratios Gallopoulos, E. Houstis, and J.R. Rice. Computer as thinker/-doer: problem-solving environments for computational science. *IEEE Computational Science and Engineering*, 1(2):11–23, 1994.
- [5] Intel® Math Kernel Library (MKL). <https://software.intel.com/en-us/mkl>. Accessed on: 2019-09-26.
- [6] Theodore H. Romer, Dennis Lee, Geoffrey M. Voelker, Alec Wolman, Wayne A. Wong, Jean-Loup Baer, Brian N. Bershad, and Henry M. Levy. The structure and performance of interpreters. In *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems - ASPLOS-VII*, volume 5, pages 150–159, New York, New York, USA, 1996. ACM Press.
- [7] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Pearson Education, Inc, 2006.
- [8] John Aycock. A brief history of just-in-time. *ACM Computing Surveys*, 35(2):97–113, 2003.
- [9] Vijay Menon and Keshav Pingali. A case for source-level transformations in MATLAB. *ACM SIGPLAN Notices*, 35(1):53–65, 1 2000.
- [10] MATLAB Execution Engine. <https://www.mathworks.com/products/matlab/matlab-execution-engine.html>. Accessed on: 2019-09-28.

- [11] Luiz De Rose, Kyle Gallivan, Efstratios Gallopoulos, B. Marsolf, and David Padua. FALCON: A MATLAB interactive restructuring compiler. In *Lecture Notes in Computer Science*, number 1448, pages 269–288. 1996.
- [12] M.J. Quinn, A. Malishevsky, and N. Seeram. Otter: bridging the gap between MATLAB and ScaLAPACK. In *Proceedings. The Seventh International Symposium on High Performance Distributed Computing (Cat. No.98TB100244)*, volume 1998-July, pages 114–121. IEEE Comput. Soc, 1998.
- [13] Joao Bispo, Pedro Pinto, Ricardo Nobre, Tiago Carvalho, Joao M. P. Cardoso, and Pedro C Diniz. The MATISSE MATLAB compiler. In *2013 11th IEEE International Conference on Industrial Informatics (INDIN)*, pages 602–608. IEEE, 7 2013.
- [14] Laurie Hendren, Jesse Doherty, Anton Dubrau, Rahul Garg, Nurudeen Lameed, Soroush Radpour, Amina Aslam, Toheed Aslam, Andrew Casey, Maxime Chevalier Boisvert, Jun Li, Clark Verbrugge, and Olivier Savary Belanger. McLAB: Enabling Programming Language, Compiler and Software Engineering Research for Matlab. *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, pages 195–196, 2011.
- [15] Xu Li and Laurie Hendren. Mc2For: A tool for automatically translating MATLAB to FORTRAN 95. In *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, pages 234–243. IEEE, 2 2014.
- [16] Vineet Kumar and Laurie Hendren. MIX10. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications - OOPSLA '14*, pages 617–636, New York, New York, USA, 2014. ACM Press.
- [17] Pramod G. Joisha and Prithviraj Banerjee. An algebraic array shape inference system for MATLAB®. *ACM Transactions on Programming Languages and Systems*, 28(5):848–907, 2006.
- [18] George Almási and David Padua. MaJIC: Compiling MATLAB for Speed and Responsiveness*. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation - PLDI '02*, page 294, New York, New York, USA, 2002. ACM Press.
- [19] Stéphane Chauveau and François Bodin. Menhir: An Environment for High Performance Matlab. *Scientific Programming*, 7(3-4):303–312, 1999.

- [20] Arun Chauhan and Ken Kennedy. Reducing and Vectorizing Procedures for Telescoping Languages. *International Journal of Parallel Programming*, 30(4):291–315, 2002.
- [21] Neil Birkbeck, Jonathan Levesque, and Jose Nelson Amaral. A Dimension Abstraction Approach to Vectorization in Matlab. In *International Symposium on Code Generation and Optimization (CGO'07)*, pages 115–130. IEEE, 3 2007.
- [22] Maxime Chevalier-Boisvert, Laurie Hendren, and Clark Verbrugge. Optimizing Matlab through Just-In-Time Specialization. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 6011 LNCS, pages 46–65. 2010.
- [23] George Almasi and David Padua. MaJIC: A Matlab just-in-time Compiler. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 2017, pages 68–81, 2001.
- [24] Ivano Azzini, Ronal Muresano, and Marco Ratto. Dragonfly: A multi-platform parallel toolbox for MATLAB/Octave. *Computer Languages, Systems and Structures*, 52:21–42, 6 2018.
- [25] Vijay Menon and Anne E Trefethen. MultiMATLAB: Integrating Matlab with high performance parallel computing. In *Proceedings of the 1997 ACM/IEEE conference on Supercomputing (CDROM) - Supercomputing '97*, volume 1, pages 1–18, New York, New York, USA, 1997. ACM Press.
- [26] Corinne Ancourt and François Irigoin. Scanning polyhedra with DO loops. *ACM SIGPLAN Notices*, 26(7):39–50, 7 1991.
- [27] Monica S. Lam, Edward E Rothberg, and Michael E Wolf. The cache performance and optimizations of blocked algorithms. *ACM SIGARCH Computer Architecture News*, 19(2):63–74, 1991.
- [28] Kathryn S. McKinley, Steve Carr, and Chau-Wen Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, 1996.
- [29] L. DeRose, Kyle Gallivan, Efstratios Gallopoulos, B. Marsolf, and David Padua. An environment for the rapid prototyping and development of numerical programs and libraries for scientific computation. *Parallel Computing*, (1370), 1994.

- [30] Luiz DeRose, Kyle Gallivan, Efstratios Gallopoulos, Bret A Marsolf, and David Padua. FALCON: An Environment for the Development of Scientific Libraries and Applications. *Proc. First International Workshop on Knowledge-Based System for the (re)Use of Program Libraries*, (November), 1995.
- [31] Patryk Kiepas, Jaroslaw Kozlak, Claude Tadonki, and Corinne Ancourt. Profile-based vectorization for MATLAB. In *Proceedings of the 5th ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming - ARRAY 2018*, pages 18–23, New York, New York, USA, 2018. ACM Press.
- [32] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65, 2009.
- [33] Ahmad Yasin. A Top-Down method for performance analysis and counters architecture. In *International Symposium on Performance Analysis of Systems and Software, ISPASS 2014*, pages 35–44, 2014.
- [34] John Cavazos, Grigori Fursin, Felix Agakov, Edwin Bonilla, Michael F.P. O’Boyle, and Olivier Temam. Rapidly Selecting Good Compiler Optimizations using Performance Counters. In *International Symposium on Code Generation and Optimization (CGO’07)*, pages 185–197. IEEE, 3 2007.
- [35] Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. End-to-End Deep Learning of Optimization Heuristics. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, volume 2017-Septe, pages 219–232. IEEE, 9 2017.
- [36] Edd Barrett, Carl Friedrich Bolz-Tereick, Rebecca Killick, Sarah Mount, and Laurence Tratt. Virtual machine warmup blows hot and cold. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–27, 10 2017.
- [37] Keith Adams, Jason Evans, Bertrand Maher, Guilherme Ottoni, Andrew Paroski, Brett Simmers, Edwin Smith, and Owen Yamauchi. The hiphop virtual machine. *ACM SIGPLAN Notices*, 49(10):777–790, 2015.
- [38] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. One VM to rule them all. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software - Onward! ’13*, pages 187–204, New York, New York, USA, 2013. ACM Press.

- [39] OpenJDK Wiki: HotSpot Internals. <https://wiki.openjdk.java.net/display/HotSpot>. Accessed on: 2019-09-30.
- [40] Michael R Jantz and Prasad A Kulkarni. Exploring single and multilevel JIT compilation policy for modern machines. *ACM Transactions on Architecture and Code Optimization*, 10(4):1–29, 2014.
- [41] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Producing wrong data without doing anything obviously wrong! *ACM SIGPLAN Notices*, 44(3):265, 2 2009.
- [42] Shruti Patil and David J. Lilja. Statistical methods for computer performance evaluation. *Wiley Interdisciplinary Reviews: Computational Statistics*, 4(1):98–106, 1 2012.
- [43] PAPI: Performance Application Programming Interface. <https://icl.utk.edu/papi/>. Accessed on: 2019-09-24.
- [44] Functions and Objects Supported for C/C++ Code Generation (MATLAB R2019b). <https://www.mathworks.com/help/coder/ug/functions-and-objects-supported-for-cc-code-generation.html>. Accessed on: 2019-09-22.
- [45] João Bispo and João M.P. Cardoso. A MATLAB subset to C compiler targeting embedded systems. *Software - Practice and Experience*, 47(2):249–272, 2 2017.
- [46] Luís Reis, João Bispo, and João M. P. Cardoso. Compiler Techniques for Efficient MATLAB to OpenCL Code Generation. In *Proceedings of the 5th International Workshop on OpenCL - IWOCCL 2017*, volume Part F1277, pages 1–2, New York, New York, USA, 2017. ACM Press.
- [47] Luiz De Rose and David Padua. A MATLAB to Fortran 90 translator and its effectiveness. In *Proceedings of the 10th international conference on Supercomputing - ICS '96*, number 1462, pages 309–316, New York, New York, USA, 1996. ACM Press.
- [48] Luiz Antonio de Rose and David Padua. Techniques for the translation of MATLAB programs into Fortran 90. *ACM Transactions on Programming Languages and Systems*, 21(2):286–323, 3 1999.
- [49] M.J. Quinn, Alexey Malishevsky, Nagajagadeswar Seelam, and Yan Zhao. Preliminary results from a parallel MATLAB compiler. In *Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing*, pages 81–87. IEEE Comput. Soc, 1998.

- [50] Stéphane Chauveau and François Bodin. Menhir: An Environment for High Performance Matlab. In *LCR '98: Selected Papers from the 4th International Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, pages 27–40. 1998.
- [51] Prithviraj Banerjee, Alok Choudhary, S Hauck, Nagaraj Shenoy, C Bachmann, M Chang, Malay Haldar, Pramod G. Joisha, A Jones, Abhay Kanhere, A Nayak, S Periyacheri, and M Walkden. MATCH: A MATLAB Compiler For Conngurable Computing Systems. Technical report, Electrical and Computer Engineering; Northwestern University, 1999.
- [52] Prithviraj Banerjee, N. Shenoy, A. Choudhary, S. Hauck, C. Bachmann, M. Haldar, Pramod G. Joisha, A. Jones, A. Kanhare, A. Nayak, S. Periyacheri, M. Walkden, and D. Zaretsky. A MATLAB compiler for distributed, heterogeneous, reconfigurable computing systems. In *Proceedings 2000 IEEE Symposium on Field-Programmable Custom Computing Machines (Cat. No.PR00871)*, pages 39–48. IEEE Comput. Soc, 2000.
- [53] Prithviraj Banerjee. An overview of a compiler for mapping MATLAB programs onto FPGAs. In *Proceedings of the 2003 conference on Asia South Pacific design automation - ASPDAC*, page 477, New York, New York, USA, 2003. ACM Press.
- [54] Pramod G. Joisha and Prithviraj Banerjee. A translator system for the MATLAB language. *Software: Practice and Experience*, 37(5):535–578, 4 2007.
- [55] Peter Jurica and van Cees Leeuwen. OMPC: an open-source MATLAB®-to-Python compiler. *Frontiers in Neuroinformatics*, 3:5, 2009.
- [56] Ashwin Prasad, Jayvant Anantpur, and R. Govindarajan. Automatic compilation of MATLAB programs for synergistic execution on heterogeneous processors. *ACM SIGPLAN Notices*, 47(6):152, 8 2012.
- [57] SILKAN eVariX/COLD. <http://www.evarix.fr/>. Accessed on: 2019-09-30.
- [58] Geir Yngve Paulsen, Jonathan Feinberg, Xing Cai, Bjorn Nordmoen, and Hans Petter Dahle. Matlab2cpp: A Matlab-to-C++ code translator. In *2016 11th System of Systems Engineering Conference (SoSE)*, pages 1–5. IEEE, 6 2016.
- [59] Geir Yngve Paulsen, Stuart Clark, Bjørn Nordmoen, Sergey Nenakhov, Aron Andersson, Xing Cai, and Hans Petter Dahle. Automated Translation of MATLAB Code to C++ with Performance and Traceability.

- In *The Eleventh International Conference on Advanced Engineering Computing and Applications in Sciences*, number c, pages 50–55, 2017.
- [60] Johannes Spazier, Steffen Christgau, and Bettina Schnor. Automatic generation of parallel C code for stencil applications written in MATLAB. *Proceedings of the 3rd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming - ARRAY 2016*, pages 47–54, 2016.
- [61] Vincent Foley-Bourgon and Laurie Hendren. Efficiently implementing the copy semantics of MATLAB’s arrays in JavaScript. In *Proceedings of the 12th Symposium on Dynamic Languages - DLS 2016*, pages 72–83, New York, New York, USA, 2016. ACM Press.
- [62] Ioannis Latifis, Karthick Parashar, Grigoris Dimitroulakos, Hans Cappelle, Christakis Lezos, Konstantinos Masselos, and Francky Catthoor. A MATLAB Vectorizing Compiler Targeting Application-Specific Instruction Set Processors. *ACM Transactions on Design Automation of Electronic Systems*, 22(2):1–28, 2017.
- [63] Pramod G. Joisha, Abhay Kanhere, Prithviraj Banerjee, Nagaraj Shenoy, and Alok Choudhary. The Design and Implementation of a Parser and Scanner for the MATLAB Language in the MATCH Compiler. Technical Report September, Center for Parallel and Distributed Computing; Northwestern University, Sheridan Road, 1999.
- [64] Pramod G. Joisha, Abhay Kanhere, Prithviraj Banerjee, U Nagaraj Shenoy, and Alok Choudhary. Handling context-sensitive syntactic issues in the design of a front-end for a MATLAB compiler. *ACM SIGAPL APL Quote Quad*, 31(3):27–40, 3 2001.
- [65] Pramod G. Joisha and Prithviraj Banerjee. Implementing an Array Shape Inference System for MATLAB Using MATHEMATICA. Technical Report October, Center for Parallel and Distributed Computing; Northwestern University, 2002.
- [66] João Bispo, Luís Reis, and João M. P. Cardoso. C and OpenCL generation from MATLAB. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing - SAC ’15*, pages 1315–1320, New York, New York, USA, 2015. ACM Press.
- [67] João M.P. Cardoso, Tiago Carvalho, José G.F. Coutinho, Wayne Luk, Ricardo Nobre, Pedro C. Diniz, and Zlatko Petrov. LARA: An aspect-oriented programming language for embedded systems. *AOSD’12 - Proceedings of the 11th Annual International Conference on Aspect Oriented Software Development*, pages 179–190, 2012.

- [68] Open Multi-Processing (OpenMP). <https://www.openmp.org/>. Accessed on: 2019-09-22.
- [69] Intel® Threading Building Blocks (TBB). <https://github.com/intel/tbb>. Accessed on: 2019-09-22.
- [70] Anton Willy Dubrau and Laurie Jane Hendren. Taming MATLAB. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications - OOPSLA '12*, page 503, New York, New York, USA, 2012. ACM Press.
- [71] Randy Allen and Ken Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, 10 1987.
- [72] John R. Allen and Ken Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann, 2001.
- [73] Vijay Menon and Keshav Pingali. High-level semantic optimization of numerical codes. In *Proceedings of the 13th international conference on Supercomputing - ICS '99*, pages 434–443, New York, New York, USA, 1999. ACM Press.
- [74] Daniel Elphick, Michael Leuschel, and Simon Cox. Partial Evaluation of MATLAB. In *GPCE '03 Proceedings of the 2nd international conference on Generative programming and component engineering*, pages 344–363, 2003.
- [75] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, 2004.
- [76] Malay Haldar, Anshuman Nayak, Abhay Kanhere, Pramod Joisha, Nagaraj Shenoy, Alok Choudhary, and Prithviraj Banerjee. Match virtual machine: An adaptive runtime system to execute MATLAB in parallel. *Proceedings of the International Conference on Parallel Processing*, pages 145–152, 2000.
- [77] Jesse Doherty, Laurie Hendren, and Soroush Radpour. Kind analysis for MATLAB. *ACM SIGPLAN Notices*, 46(10):99, 10 2011.
- [78] Reinhard Wilhelm, Mooly Sagiv, and Thomas Reps. Shape Analysis. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 1781, pages 1–17. 2000.

- [79] MAGICA: A Type Inference Engine for MATLAB. <http://www.ece.northwestern.edu/cpdc/pjoisha/MAGICA/>. Accessed on: 2019-09-29.
- [80] Pramod G. Joisha and Prithviraj Banerjee. The MAGICA Type Inference Engine for MATLAB ®. In *Compiler Construction. CC 2003. Lecture Notes in Computer Science*, pages 121–125. Springer, Berlin, Heidelberg, vol 2622 edition, 2003.
- [81] Pramod G. Joisha. *A type inference system for MATLAB with applications to code optimization*. PhD thesis, Northwestern University, 2003.
- [82] Shankar Ramaswamy, E.W. Hodges, and P Banerjee. Compiling MATLAB programs to ScaLAPACK: exploiting task and data parallelism. In *Proceedings of International Conference on Parallel Processing*, pages 613–619. IEEE Comput. Soc. Press, 1996.
- [83] Andreas Abel and Jan Reineke. uops.info: Characterizing Latency, Throughput, and Port Usage of Instructions on Intel Microarchitectures. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS '19*, pages 673–686, New York, New York, USA, 2019. ACM Press.
- [84] John L. Hennessy and David Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2017.
- [85] David Callahan, John Cocke, and Ken Kennedy. Estimating interlock and improving balance for pipelined architectures. *Journal of Parallel and Distributed Computing*, 5(4):334–358, 8 1988.
- [86] Steve Carr, Kathryn S. McKinley, and Chau-Wen Tseng. Compiler optimizations for improving data locality. *ACM SIGOPS Operating Systems Review*, 28(5):252–262, 1994.
- [87] Aleksandar Ilic, Frederico Pratas, and Leonel Sousa. Cache-aware roofline model: Upgrading the loft. *IEEE Computer Architecture Letters*, 13(1):21–24, 2014.
- [88] Aleksandar Ilic, Frederico Pratas, and Leonel Sousa. Beyond the roofline: Cache-aware power and energy-efficiency modeling for multi-cores. *IEEE Transactions on Computers*, 66(1):52–58, 2017.
- [89] Victoria Caparros Cabezas and Markus Puschel. Extending the roofline model: Bottleneck analysis with microarchitectural constraints. In *IISWC 2014 - IEEE International Symposium on Workload Characterization*, pages 222–231, 2014.

- [90] Georg Hager, Jan Treibig, Johannes Habich, and Gerhard Wellein. Exploring performance and power properties of modern multi-core chips via simple machine models. *Concurrency and Computation: Practice and Experience*, 28(2):189–210, 2 2016.
- [91] Brinkley Sprunt. The basics of performance-monitoring hardware. *IEEE Micro*, 22(4):64–71, 2002.
- [92] Stéphane Eranian. What can performance counters do for memory subsystem analysis? In *Proceedings of the 2008 ACM SIGPLAN workshop on Memory systems performance and correctness held in conjunction with the Thirteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '08) - MSPC '08*, page 26, New York, New York, USA, 2008. ACM Press.
- [93] OProfile. <https://oprofile.sourceforge.io/>. Accessed on: 2019-09-24.
- [94] perfmon2. <http://perfmon2.sourceforge.net/>. Accessed on: 2019-09-24.
- [95] Dan Terpstra, Heike Jagode, Haihang You, and Jack Dongarra. Collecting Performance Data with PAPI-C. In *Tools for High Performance Computing 2009*, pages 157–173, 2010.
- [96] Tiptop. <http://tiptop.gforge.inria.fr/>. Accessed on: 2019-09-24.
- [97] Erven Rohou. Tiptop: Hardware Performance Counters for the Masses. Technical report, Inria Rennes – Bretagne Atlantique, 2011.
- [98] Erven Rohou. Tiptop: Hardware performance counters for the masses. *Proceedings of the International Conference on Parallel Processing Workshops*, pages 404–413, 2012.
- [99] Intel® VTune. <https://software.intel.com/en-us/vtune>. Accessed on: 2019-09-24.
- [100] V Weaver and Jack Dongarra. Can hardware performance counters produce expected, deterministic results. *Proceedings of Third Workshop on Functionality of Hardware Performance Monitoring*, 2010.
- [101] Vincent M Weaver and Sally A. McKee. Can hardware performance counters be trusted? In *International Symposium on Workload Characterization, IISWC 2008*, pages 141–150, 2008.
- [102] Dmitrijs Zaparanuks, Milan Jovic, and Matthias Hauswirth. Accuracy of performance counter measurements. In *International Symposium on Performance Analysis of Systems and Software, ISPASS 2009*, pages 23–32, 2009.

- [103] Florian T. Schneider, Mathias Payer, and Thomas R. Gross. On-line optimizations driven by hardware performance monitoring. *ACM SIGPLAN Notices*, 42(6):373, 2007.
- [104] Lieven Eeckhout, Andy Georges, and Koen De Bosschere. How java programs interact with virtual machines at the microarchitectural level. *ACM SIGPLAN Notices*, 38(11):169, 2003.
- [105] Peter F Sweeney, Matthias Hauswirth, Brendon Cahoon, Perry Cheng, Amer Diwan, David Grove, and Michael Hind. Using hardware performance monitors to understand the behavior of java applications. *Virtual Machine Research And Technology Symposium, VM 2004*, page 5, 2004.
- [106] Timothy Sherwood, Brad Calder, and San Diego. Time Varying Behavior of Programs. Technical report, Department of Computer Science and Engineering; University of California, San Diego, 1999.
- [107] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS 2002*, volume 36, page 45, 2002.
- [108] T. Sherwood, E. Perelman, Greg Hamerly, S. Sair, and Brad Calder. Discovering and exploiting program phases. *IEEE Micro*, 23(6):84–93, 2003.
- [109] E. Duesterwald, C. Cascaval, and Sandhya Dwarkadas. Characterizing and predicting program behavior and its variability. In *Parallel Architectures and Compilation Techniques, PACT 2003*, pages 220–231, 2003.
- [110] Matthias Hauswirth, Peter F. Sweeney, Amer Diwan, and Michael Hind. Vertical profiling: Understanding the Behavior of Object-Oriented Applications. In *Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2004*, volume 39, page 251, 2004.
- [111] Matthias Hauswirth, Amer Diwan, Peter F Sweeney, and Michael C Mozer. Automating vertical profiling. *ACM SIGPLAN Notices*, 40(10):281, 2006.
- [112] Matthias Hauswirth, Peter F Sweeney, and Amer Diwan. Temporal vertical profiling. *Software - Practice and Experience*, 40(8):627–654, 2010.
- [113] M. Anton Ertl and David Gregg. The Behavior of Efficient Virtual Machine Interpreters on Modern Architectures. In *Lecture Notes in*

Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), volume 2150, pages 403–413. 2 2001.

- [114] M Anton Ertl and David Gregg. The structure and performance of efficient interpreters. *Journal of Instruction-Level Parallelism*, 5:1–25, 2003.
- [115] Stefan Brunthaler. Virtual-Machine Abstraction and Optimization Techniques. *Electronic Notes in Theoretical Computer Science*, 253(5):3–14, 12 2009.
- [116] Erven Rohou, Bharath Narasimha Swamy, and Andre Seznec. Branch prediction and the performance of interpreters – Don’t trust folklore. In *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 103–114. IEEE, 2 2015.
- [117] Gergő Barany. Python Interpreter Performance Deconstructed. In *Proceedings of the Workshop on Dynamic Languages and Applications - Dyla’14*, pages 1–9, New York, New York, USA, 2014. ACM Press.
- [118] Vincenza Carchiolo, Michele Malgeri, Giuseppe Mangioni, and Vincenzo Nicosia. Evaluating the Dynamic Behaviour of Python applications. In *ACSC ’09 Proceedings of the Thirty-Second Australasian Conference on Computer Science*, pages 19–28, 2009.
- [119] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *International Symposium on Code Generation and Optimization, CGO 2003*, pages 265–275, 2003.
- [120] Chi-keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Programming Language Design and Implementation, PLDI 2005*, page 190. ACM Press, 2005.
- [121] Rafael H. Saavedra and Alan Jay Smith. Analysis of benchmark characteristics and benchmark performance prediction. *ACM Transactions on Computer Systems*, 14(4):344–384, 1996.
- [122] Shirley V Moore. A comparison of counting and sampling modes of using performance monitoring hardware. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 2330 LNCS, pages 904–912, 2002.

- [123] Jack Doweck, Wen-Fu Kao, Allen Kuan-yu Lu, Julius Mandelblat, Anirudha Rahatekar, Lihu Rappoport, Efraim Rotem, Ahmad Yasin, and Adi Yoaz. Inside 6th-Generation Intel Core: New Microarchitecture Code-Named Skylake. *IEEE Micro*, 37(2):52–62, 2017.
- [124] Todd Mytkowicz, Peter Sweeney, Matthias Hauswirth, Amer Diwan, Peter F Sweeney, and Unisich Amer Diwan. Observer Effect and Measurement Bias in Performance Analysis. Technical Report June, University of Colorado at Boulder, 2008.
- [125] Daniel Molka, Robert Schöne, Daniel Hackenberg, and Wolfgang E Nagel. Detecting Memory-Boundedness with Hardware Performance Counters. volume 17, pages 27–38, 2017.
- [126] John D McCalpin. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, (May):19–25, 1995.
- [127] The Bandwidth Benchmark: Extended Stream. <https://github.com/RRZE-HPC/TheBandwidthBenchmark>. Accessed on: 2019-09-19.
- [128] Jerry L Hintze and Ray D Nelson. Violin plots: A box plot-density trace synergism. *American Statistician*, 52(2):181–184, 1998.
- [129] Philip Samuel Abrams. *An APL Machine*. PhD thesis, Stanford University, 1970.
- [130] Mike Shaver, David Mandelin, Jason Orendorff, Michael Bebenita, Brendan Eich, Michael Franz, Rick Reitmaier, David Anderson, Edwin W. Smith, Boris Zbarsky, Mason Chang, Graydon Hoare, Mohammad R. Haghighat, Blake Kaplan, Jesse Ruderman, and Andreas Gal. Trace-based just-in-time type specialization for dynamic languages. *ACM SIGPLAN Notices*, 44(6):465, 2009.
- [131] Jacob Brock, Chen Ding, Xiaoran Xu, and Yan Zhang. PAYJIT: space-optimal JIT compilation and its practical implementation. In *Proceedings of the 27th International Conference on Compiler Construction - CC 2018*, volume 18, pages 71–81, New York, New York, USA, 2018. ACM Press.
- [132] Intel. Intel ® 64 and IA-32 Architectures Software Developer’s Manual. Technical report, 2019.
- [133] Intel® Processor Event Reference. <https://download.01.org/perfmon/index/>. Accessed on: 2019-09-13.

- [134] FFTW: Fastest Fourier Transform in the West. <http://www.fftw.org/>. Accessed on: 2019-09-26.
- [135] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. In *Proceedings of the IEEE*, volume 93, pages 216–231, 2 2005.
- [136] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. Julia: A fresh approach to numerical computing. *SIAM Review*, 59(1):65–98, 2017.
- [137] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. Tracing the Meta-Level: PyPy’s Tracing JIT Compiler. In *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems - ICOOLPS ’09*, pages 18–25, New York, New York, USA, 2009. ACM Press.
- [138] PyPy: Tracing JIT compiler for Python. <https://pypy.org/>. Accessed on: 2019-09-27.
- [139] Samuel Larsen and Saman Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. *SIGPLAN Notices (ACM Special Interest Group on Programming Languages)*, 35(5):145–156, 2000.
- [140] Michael Wolfe. Vector optimization vs vectorization. *Journal of Parallel and Distributed Computing*, 5(5):551–567, 1988.
- [141] D. Callahan, J. Dongarra, and D. Levine. Vectorizing compilers: a test suite and results. In *Proceedings. SUPERCOMPUTING ’88*, pages 98–105. IEEE Comput. Soc. Press, 1988.
- [142] Saeed Maleki, Yaoqing Gao, María J. Garzarán, Tommy Wong, and David A. Padua. An evaluation of vectorizing compilers. *Parallel Architectures and Compilation Techniques, PACT 2011*, 7:372–382, 2011.
- [143] Randolph G. Scarborough and Harwood G. Kolsky. A vectorizing Fortran compiler. *IBM Journal of Research and Development*, 30(2):163–171, 3 1986.
- [144] J. R. Allen, Ken Kennedy, Carrie Porterfield, and Joe Warren. Conversion of control dependence to data dependence. In *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages - POPL ’83*, pages 177–189, New York, New York, USA, 2003. ACM Press.

- [145] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous java performance evaluation. *ACM SIGPLAN Notices*, 42(10):57, 10 2007.
- [146] Rose Compiler: Program Analysis and Transformation. <http://rosecompiler.org/>. Accessed on: 2019-09-30.
- [147] PIPS: Automatic Parallelizer and Code Transformation Framework. <https://pips4u.org/>. Accessed on: 2019-09-30.
- [148] Cetus: A Source-to-Source Compiler Infrastructure for C Programs. <https://engineering.purdue.edu/Cetus/>. Accessed on: 2019-09-30.
- [149] Hansang Bae, Dheya Mustafa, Jae-Woo Lee, Aurangzeb, Hao Lin, Chirag Dave, Rudolf Eigenmann, and Samuel P. Midkiff. The Cetus Source-to-Source Compiler Infrastructure: Overview and Evaluation. *International Journal of Parallel Programming*, 41(6):753–767, 12 2013.
- [150] Frances E. Allen. Control flow analysis. *ACM SIGPLAN Notices*, 5(7):1–19, 7 1970.
- [151] Frances E. Allen and John Cocke. A program data flow analysis procedure. *Communications of the ACM*, 19(3):137, 1976.
- [152] David J. Kuck, R. H. Kuhn, David Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '81*, pages 207–218, New York, New York, USA, 1981. ACM Press.
- [153] Polly: LLVM Framework for High-Level Loop and Data-Locality Optimizations. <https://polly.llvm.org/>. Accessed on: 2019-09-14.
- [154] Terence Parr. *Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages*. Pragmatic Bookshelf, 2009.
- [155] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation - PLDI '13*, page 519, New York, New York, USA, 2013. ACM Press.
- [156] Andrew Casey, Soroush Radpour, Olivier Savary Belanger, Laurie Hendren, Clark Verbrugge, Jun Li, Jesse Doherty, Maxime Chevalier-Boisvert, Toheed Aslam, Anton Dubrau, Nurudeen Lameed, Amina Aslam, and Rahul Garg. McLab: an extensible compiler toolkit for

- MATLAB and related languages. *Proceedings of the Third C* Conference on Computer Science and Software Engineering - C3S2E '10*, pages 114–117, 2010.
- [157] Amina Aslam and Laurie Hendren. McFLAT: A profile-based framework for MATLAB loop analysis and transformations. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 6548 LNCS, pages 1–15, 2011.
 - [158] Andrew Bodzay and Laurie Hendren. AspectMatlab++: annotations, types, and aspects for scientists. In *Proceedings of the 14th International Conference on Modularity - MODULARITY 2015*, pages 41–54, New York, New York, USA, 2015. ACM Press.
 - [159] M. Sharir. A strong-connectivity algorithm and its applications in data flow analysis. *Computers & Mathematics with Applications*, 7(1):67–72, 1981.
 - [160] J. J. Dongarra and A. R. Hinds. Unrolling Loops in FORTRAN. *Software: Practice and Experience*, 9(3):219–226, 1979.
 - [161] Laurie Hendren. Typing aspects for MATLAB. In *Proceedings of the sixth annual workshop on Domain-specific aspect languages - DSAL '11*, page 13, New York, New York, USA, 2011. ACM Press.
 - [162] Krun: High fidelity benchmark runner. <https://soft-dev.org/src/krun/>. Accessed on: 2019-09-30.
 - [163] Aleksander Maricq, Dmitry Duplyakin, Ryan Stutsman, Robert Ricci, Carlos Maltzahn, and Ivo Jimenez. Taming Performance Variability. *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation - OSDI'18*, pages 409–425, 2018.
 - [164] perf: Linux profiling with performance counters. https://perf.wiki.kernel.org/index.php/Main_Page. Accessed on: 2019-09-30.
 - [165] April W. Wade, Prasad A. Kulkarni, and Michael R. Jantz. AOT vs. JIT: impact of profile data on code quality. *ACM SIGPLAN Notices*, 52(4):1–10, 2017.
 - [166] Matthew Arnold, S.J. Fink, David Grove, Michael Hind, and P.F. Sweeney. A Survey of Adaptive Optimization in Virtual Machines. *Proceedings of the IEEE*, 93(2):449–466, 2 2005.

RÉSUMÉ

MATLAB est un environnement informatique doté d'un langage de programmation simple et d'une vaste bibliothèque de fonctions couramment utilisées en science et ingénierie (CSE) pour le prototypage rapide. Cependant, certaines caractéristiques de son environnement, comme son langage dynamique ou son style de programmation interactif, affectent la rapidité d'exécution des programmes. Les approches actuelles d'amélioration des programmes MATLAB traduisent le code dans des langages statiques plus rapides comme C ou Fortran, ou bien appliquent systématiquement des transformations de code au programme MATLAB sans considérer leur impact sur les performances. Dans cette thèse, nous comblons cette lacune en développant des techniques d'analyse et de transformation de code des programmes MATLAB afin d'augmenter leur performance. Plus précisément, nous analysons et modélisons le comportement d'un environnement MATLAB black-box uniquement en mesurant l'exécution caractéristique des programmes sur CPU. À partir des données obtenues, nous formalisons un modèle statique qui prédit le type et l'ordonnancement des instructions programmées lors de l'exécution par le compilateur Just-In-Time (JIT). Ce modèle nous permet de proposer plusieurs transformations de code qui améliorent les performances des programmes MATLAB en influençant la façon dont le compilateur JIT génère le code machine. Les résultats obtenus démontrent les avantages pratiques de la méthodologie présentée.

MOTS CLÉS

Optimisation du programme, Analyse de performance, Compteurs de performance, MATLAB, Modèle d'exécution, Transformation de code

ABSTRACT

MATLAB is a computing environment with an easy programming language and a vast library of functions commonly used in Computation Science and Engineering (CSE) for fast prototyping. However, some features of its environment, such as its dynamic language or interactive style of programming affect how fast the programs can execute. Current approaches to improve MATLAB programs either translate the code to faster static languages like C or Fortran, or apply code transformations to MATLAB code systematically without considering their impact on the performance. In this thesis, we fill this gap by developing techniques for the analysis and code transformation of MATLAB programs in order to improve their performance. More precisely, we analyse and model the behaviour of the black-box MATLAB environment by measuring the execution characteristics of programs on CPU. From the resulting data, we formalise a static model which predicts the type and order of instructions scheduled by the Just-In-Time (JIT) compiler. This model allows us to propose several code transformations which increase the performance of MATLAB programs by influencing how the JIT compiler generates the machine code. The obtained results demonstrate the practical benefits of the presented methodology.

KEYWORDS

Program optimisation, Performance analysis, Performance counters, MATLAB, Execution model, Code transformation