Co-design of FPGA Implementations for Model Predictive Control

Bulat Khusainov

Department of Electrical and Electronic Engineering

Imperial College London

This dissertation is submitted for the degree of

Doctor of Philosophy

January 2018

I would like to dedicate this thesis to my parents.

Abstract

Model Predictive Control (MPC) is an advanced control method that is capable of explicit performance optimization, systematic constraint handling and dealing with nonlinearities in a natural way. The necessity of solving an optimization problem at each sampling instant makes MPC a computationally demanding technique, especially when applied to fast dynamical systems. This thesis is concerned with developing efficient hardware implementations of linear and nonlinear model predictive controllers.

In the first part of this thesis, a software toolchain for quick prototyping of embedded optimization algorithms on Field-Programmable Gate Arrays (FPGAs) is presented. The toolchain consists of two software tools: SPLIT and Protoip. SPLIT is capable of generating CPU and FPGA-oriented C code for embedded optimization using operator splitting methods. The generated code can be automatically deployed and tested on an embedded platform using a new release of Protoip, a software tool for quick prototyping of optimization algorithm on CPUs, FPGAs and heterogeneous platforms that incorporate both general-purpose processors and reconfigurable logic.

The second part presents a framework for implementation of nonlinear model predictive control on a heterogeneous computing platform. Splitting the computational workload between a general-purpose CPU and an FPGA allows exploiting the strengths of each computational subsystem and trading off control performance against reconfigurable logic usage. A new method for scheduling sparse matrix-vector multiplication within the proposed implementation enables significant improvements in terms of memory and computational resources usage.

The third part of this thesis presents an application of systematic optimization to the co-design of MPC software and computational hardware. A procedure for formulating the MPC design problem as a multi-objective optimization problem is described. Finally, two test cases including CPU and FPGA-based implementations of predictive controllers are considered.

Declaration of Originality

I hereby certify that all material in this dissertation is my own work, and, where appropriate, contributions from other people have been properly acknowledged.

Bulat Khusainov

January 2018

Copyright Declaration

The copyright of this thesis rests with the author and is made available under a Creative Commons Attribution Non-Commercial No Derivatives licence. Researchers are free to copy, distribute or transmit the thesis on the condition that they attribute it, that they do not use it for commercial purposes and that they do not alter, transform or build upon it. For any reuse or redistribution, researchers must make clear to others the licence terms of this work.

Acknowledgements

I would like to express my gratitude to both of my supervisors for their continuous support and guidance. To Dr Eric Kerrigan for giving me research freedom, proposing interesting ideas, sharing his research tips and being a patient mentor. To Prof. George Constantinides for valuable discussions, constructive feedback and inculcating goal orientedness.

I am certainly grateful to my Imperial College colleagues, who were always supportive: Andrea Suardi, Luxin Zhang, Andrea Picciau, Omar Faqir, Yuanbo Nie, Junyi Liu, Ankur Majumdar, Hossein Ameli, Yunjie Gu and Husni Rois Ali. I would also like to thank Michelle Hennessy-Hammond and Wiesia Hsissen for the administrative support.

It was a pleasure to be a part of the TEMPO project, where I had a chance to meet like-minded fellows and establish research collaborations with Harsh Shukla and Deepak Ingole. Special thanks go to Prof. Moritz Diehl, Andrea Zanelli, Rien Quirynen and Robin Verschueren for hosting me at Systems Control and Optimization Laboratory at the University of Freiburg.

I would also like to thank our lunch club members: Pedro Ramirez, Himadri Das, Tiago Rodrigues, Javier Pereda and Giuseppe Laudani. We had I great time together, I will miss it.

Many thanks to my parents, sister Alsu and brother-in-law Irek for their crucial support during the hardest moments of my PhD studies.

Finally, I would like to acknowledge the financial support from the People Programme (Marie Curie Actions) of the European Union's Seventh Framework Programme (FP7/2007-2013) under REA grant agreement no 607957 (TEMPO).

Table of contents

Abbreviations **17**

List of figures **21**

List of tables **25**

1 Introduction **27**

1.1 Motivation and objectives 27

1.2 Optimal control: Brief literature review 28

1.3 Outline of the thesis 31

1.4 Contributions of the thesis 31

1.5 List of publications 32

1.5.1 Conference papers 32

1.5.2 Other conference talks 33

2 Computer Architectures for Online Optimization and Corresponding Programming Tools **35**

2.1 Computing platforms 35

2.1.1 General-purpose Central Processing Units (CPUs) 35

2.1.2 Microcontrollers 37

2.1.3 Graphical Processing Units (GPUs) 38

2.1.4 Programmable Logic Controllers (PLCs) 38

2.1.5 Field-Programmable Gate Arrays (FPGAs) 39

2.1.6 Heterogeneous processors 44

2.2 Programming languages and tools for FPGAs and heterogeneous computers . . . 45

2.3 Summary 47

3 Hardware Code Generation for Embedded Optimization with Operator Splitting Methods 49

3.1 SPLIT code generator 50

3.1.1 Introduction 50

3.1.2 Embedded optimization on FPGAs: existing work 51

3.1.3 Problem formulation 51

3.1.4 Splitting algorithms 54

3.1.5 FPGA code generation 57

3.2 Protoip 62

3.2.1 Existing features 62

3.2.2 A new release 63

3.3 Numerical results 66

3.4 Summary and future work 70

4 Nonlinear Predictive Control on a Heterogeneous Computing Platform 73

4.1 Hardware acceleration of NMPC: main challenges 73

4.2 Optimal control problem formulation and temporal discretization 74

4.3 Algorithm and implementation details 77

4.3.1 Primal-dual interior-point algorithm 77

4.3.2 Proposed implementation 79

4.3.3 Limitations of the proposed implementation 83

4.4 Experimental setup and simulation results 85

4.4.1 Experimental setup 85

4.4.2 Simulation results 87

4.5 Summary and future work 90

5 Automatic Software and Computing Hardware Co-design for Predictive Control 91

5.1 Introduction and related work 91

5.2 Formulating the design problem as an optimization problem 93

5.2.1 Design objectives and constraints 93

5.2.2 Design parameters . . . 95
5.2.3 The resulting optimization problem . . . 98
5.3 Derivative-free multi-objective optimization . . . 99
5.3.1 Problem statement . . . 99
5.3.2 Review of algorithms for derivative-free multi-objective optimization . . . 100
5.4 Case studies . . . 103
5.4.1 Fast gradient-based controller for a mass-spring-damper system - CPU-only implementation . . . 103
5.4.2 Fast gradient-based controller for a mass-spring-damper system - FPGA implementation . . . 109
5.5 Summary and future work . . . 113

6 Conclusions and Future Work 115
6.1 Contributions . . . 115
6.1.1 Automatic hardware code generation and testing for embedded optimization 115
6.1.2 Nonlinear MPC on a heterogeneous computer . . . 116
6.1.3 Automatic co-design of software and computational hardware for MPC . 116
6.2 Impact . . . 117
6.3 Future work . . . 117
6.3.1 Further automation of MPC design . . . 117
6.3.2 Energy- and time-aware algorithms . . . 118

References 119

Appendix A Experimental setup data for Chapter 5 127

Appendix B Remaining details for Chapter 4 129
B.1 Remaining details for Algorithm 4 . . . 129
B.2 Synthesis results . . . 129

Appendix C Optimal Scheduling for Sparse Matrix-Vector Multiplication on an FPGA 131
C.1 Introduction . . . 131
C.2 Problem Statement . . . 132

C.3 Scheduling Algorithm 135
C.3.1 A relaxed optimization problem for (C.5) 135
C.3.2 Scheduling procedure 137
C.4 Numerical example 139
C.5 Conclusions 140

Abbreviations

ADMM Alternating Direction Method of Multipliers

ADC Analogue to Digital Converter

AMA Alternating Minimization Algorithm

BFGS Broyden-Fletcher-Goldfarb-Shanno

BRAM Block RAM

CLB Configurable Logic Block

CMOS Complementary Metal-Oxide-Semiconductor

CPU Central Processing Unit

DDR Double Data Rate (memory)

DMS Direct Multisearch

DSM Direct Search Method

DSP Digital Signal Processor

FF Flip-Flop

FPGA Field-Programmable Gate Array

GPGPU General-Purpose GPU

GPU Graphical Processing Unit

HDL Hardware Description Language

HJB Hamilton-Jacobi-Bellman (equation)

HLS High Level Synthesis

KKT Karush-Kuhn-Tucker (conditions)

LHS Latin Hypercube Sampling

LQR Linear-Quadratic Regulator

LUT Look Up Table

MAC Multiply-Accumulate

MADS Mesh Adaptive Direct Search

MINRES Minimal Residual (algorithm)

MOO Multi-Objective optimization

MPC Model Predictive Control

NMPC Nonlinear MPC

NLP Nonlinear Programming

OCP Optimal Control Problem

ODE Ordinary Differential Equation

PCI Peripheral Component Interconnect

PDA Prima-Dual Algorithm

PLC Programmable Logic Controller

QP Quadratic Programming

RAM Random Access Memory

ROM Read-Only Memory

SoC System-on-a-Chip

TTL Transistor-Transistor Logic

UDP User Datagram Protocol

VHDL VHSIC Hardware Description Language

VHSIC Very High Speed Integrated Circuit

List of figures

2.1 CPU architecture. 36

2.2 FPGA architecture. 39

2.3 Data-level parallelism for vector-vector addition. 40

2.4 Unpipelined and pipelined implementations of an abstract algorithm that consists the sequential operations: f_1, f_2 and f_3. 41

2.5 Latency (execution time) of a fast gradient optimization-based controller, FPGA resource usage and closed-loop system performance as a function of number of fraction bits. The number of integer bits is fixed to 8. A lower closed-loop cost implies better performance. 43

2.6 FPGA resource usage for different operations. Data is obtained using vendors synthesis tool, Vivado HLS. 44

2.7 Zynq SoC architecture. 45

2.8 FPGA programming languages. 45

3.1 Toolchain design flow. 50

3.2 Dense matrix-vector multiplication with an adder tree. N_l is the dimension of the square matrix. 59

3.3 Dense matrix-vector multiplication based on multiply-accumulate units. 60

3.4 Time vs resource usage trade-off for matrix-vector multiplication implementations with SPLIT. Points are connected to improve visibility. 61

3.5 Protoip FPGA design flow. 63

3.6 Processor-in-the-loop test of FPGA implementations. Logos indicate software tools used by Protoip in the underlying levels. 63

3.7 Protoip design flow for heterogeneous implementations. 64
3.8 Communication between processing system and programmable logic with Protoip. 65
3.9 Mass-spring system. 66
3.10 Time and resource scaling of software, heterogeneous and hardware implementations. Timing results are presented for 40 iterations of the AMA algorithm. . . . 69
3.11 Trading off resources usage against latency by varying the number of MAC units. 70

4.1 Structure of the KKT matrix assosiated with the problem (4.5). H_i and ∇F_i denote Hessians and Jacobians of the corresponding sampling instant in the prediction horizon. 79
4.2 Algorithm 4 flow diagram with MINRES solver. 80
4.3 Memory requirements for the storing matrix on Figure 4.1. ODE model and OCP parameters are presented in Section 4.4, trapezoidal method is used for ODE integration. Details on dense band implementation can be found in [17]. Actual memory savings may vary depending on the memory block size of a particular computing platform. 81
4.4 An example of a sparse matrix (a) and the corresponding circuit (b) for matrix vector multiplication. 82
4.5 The impact of scheduling on computation time of a sparse matrix-vector multiplication (for each gray block in Figure 4.1). It is assumed that the latency for the addition and multiplication units are 6 and 5 clock cycles, respectively. The number of nonzero elements is 161, which is the case for the example described in Section 4.4. 83
4.6 Trading off computational time against processor resource usage for matrix-vector multiplication (Figure 4.1). Each circle represents a design with a certain degree of parallelism. The floating point unit is either floating point addition or multiplication. For the considered test case, $N = 20$ and the ODE model is given in Section 4.4. It is assumed that the latency of the addition and multiplication units are 6 and 5 clock cycles, respectively. Details on the existing implementation can be found in Boland and Constantinides [17]. 84
4.7 A schematic drawing of the gantry crane setup. 85

4.8 Trading off algorithm execution time against resource usage by splitting the workload between software and hardware. For all implementations $N = 10$. 88
4.9 The impact of horizon length on FPGA resource usage with HG_3 implementation. 88
4.10 Processor-in-the-loop simulation results with HG_3 implementation. $N = 10$, $P = 10$. 89

5.1 Mass-spring-damper system. 103
5.2 CPU in the loop test setup. 105
5.3 The impact of horizon length and sampling time on Hessian condition number. . 107
5.4 Pareto frontier approximation for CPU implementations of the FGM: BiMADS vs LHS. 108
5.5 Hypervolume profiles for CPU implementations of the FGM: BiMADS vs LHS. . 109
5.6 FPGA in the loop test setup. 109
5.7 Pareto frontier approximation for FPGA implementations of the FGM: BiMADS vs LHS. 112
5.8 Hypervolume profiles for FPGA implementations of the FGM: BiMADS vs LHS. 113

C.1 FPGA architecture for sparse matrix-vector multiplication with a single MAC unit. 134
C.2 An example of an incomplete schedule for the problem (C.1). 137
C.3 Algorithm 6 intermediate steps for the problem (C.1). 139
C.4 Sparsity pattern for the matrices that arise from optimal control, signal processing and differential equation-constrained optimization. 139
C.5 The impact of scheduling on computation time of a sparse matrix-vector multiplication (each gray block on Figure C.4) for the considered example. It is assumed that the latency for the addition and multiplication units are 6 and 5 clock cycles, respectively. The Figure is taken from [51]. 141

List of tables

3.1 Benchmark problems 67

3.2 FPGA implementations of MPC for mass-spring systems. 71

5.1 Pareto frontier approximation identified by BiMADS for CPU implementations of FGM (corresponds to Figure 5.4). Designs are sorted according to their settling times in ascending order. 108

5.2 Pareto-frontier approximation identified by BiMADS for FPGA implementations of FGM (corresponds to Figure 5.7). Designs are sorted according to their settling times in ascending order. 113

B.1 Algorithm execution times with heterogeneous implementations. For all considered implementations the CPU clock rate is 667 MHz and the FPGA clock frequency is 166 MHz. $T_s = 100$ ms, $n_{iter} = 15$, t is the algorithm execution time. 130

Chapter 1

Introduction

1.1 Motivation and objectives

Model Predictive Control (MPC) is an advanced control method that uses a mathematical model of the plant in order to predict the behaviour of the system and derive the optimal input sequence. The key outstanding features of MPC involve explicit performance optimization, systematic constraint handling and the inherent capability of dealing with nonlinearities [77].

At each sampling instant an MPC controller takes a measurement from the plant, solves an optimization problem and closes the loop by applying a part of the optimal input trajectory. With the explicit MPC approach the optimal input can be precomputed offline and expressed as a function of the current state for all possible feasible regions [14]. However, due to exponential growth of the complexity of the explicit function description with respect to the problem size, explicit MPC is often inapplicable for medium and large scale control problems. Tackling such problems requires solving Optimal Control Problems (OCPs) online, while the controlled plant is evolving. This has been the main bottleneck that historically prevented more significant expansion of MPC, especially in relation to fast dynamical systems.

Conventional approaches to overcome this limitation are based on improving MPC controllers on a software level, e.g. by using efficient optimization problem formulations and tailored optimization solvers. In addition to software improvements, recent developments in computational and communication technologies have increased capabilities of embedded real-time platforms, which potentially increases the scope of application of predictive control. However, most MPC algorithms are designed at a high level of abstraction without regard to the intended hardware platform. This

decoupled approach usually leads to inefficient utilization of available computational resources [48]. An alternative approach is based on co-design of software and computational hardware, which implies tailoring software for hardware and vice versa. Exploiting the coupling between software and hardware potentially allows achieving the best possible performance for a given amount of available computational resources. However, improvement of the closed-loop performance cannot be considered as the only design objective. In practice, there is a trade-off between performance and computational resource usage. Resources that are required to perform computations include time, energy and space, which are also the functions of the MPC algorithm and hardware platform. Instead of looking for one optimal design (which often does not exist due to conflicts between objectives) engineers might benefit from considering the whole series of Pareto optimal designs, i.e. designs that cannot be improved in terms of one objective without worsening at least one of the other objectives.

This thesis is focused on developing efficient hardware implementations for linear and nonlinear model predictive control. Several optimization algorithms including first- and second-order methods will be considered. Systematic optimization techniques will be used to solve offline scheduling problems and for parameter optimization. The distinguishing point of this work is joint optimization of software and hardware parameters and using multi-objective optimization for systematic exploration of the design trade-offs.

1.2 Optimal control: Brief literature review

Consider the following optimal control problem with a finite prediction horizon T:

$$\underset{u,x}{\text{minimize}} \quad \int_0^T L(x(t),u(t))dt + E(x(T)) \tag{1.1a}$$

$$\text{subject to} \quad x(0) = \hat{x} \tag{1.1b}$$

$$\dot{x}(t) = f_c(x(t),u(t)), \quad \forall t \in [0,T] \tag{1.1c}$$

$$g_l \leq g(x(t),u(t)) \leq g_u, \quad \forall t \in [0,T] \tag{1.1d}$$

$$\phi_l \leq \phi(x(T)) \leq \phi_u \tag{1.1e}$$

where $x(t) \in \mathbb{R}^n$ and $u(t) \in \mathbb{R}^m$ are the states and inputs at time t. The objective function is composed of the intermediate cost function $L : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}$ and the terminal cost $E : \mathbb{R}^n \to \mathbb{R}$.

System dynamics are captured by (1.1b) and (1.1c), path and terminal constraints are reflected by (1.1d) and (1.1e).

There are several analytical and numerical approaches for solving (1.1) presented in the literature:

- *Hamilton–Jacobi–Bellman (HJB) equation*-based approach [13]. The HJB equation, which is a partial differential equation (PDE), is a necessary and sufficient condition for optimality (1.1).
- *Pontryagin's maximum principle* [52], which is based on calculus of variations, provides necessary conditions for optimality (1.1).
- *Direct methods* [15] approach the problem (1.1) by discretizing control inputs and solving the resulting finite-dimensional optimization problem. MPC is based on solving OCP problems with direct methods.

In contrast to their competitors, direct methods provide flexibility in incorporating path constraints and scale well with the problem size. However, this is the most computationally intensive approach, since the resulting optimization problem is solved in real-time, when the plant under control is evolving. The family of direct methods involves:

- Shooting-based methods
  - Single shooting
  - Multiple shooting
- Direct transcription algorithms

The difference between the above methods comes from the way how they discretize the optimal control problem (1.1). The common feature of shooting methods is decoupling the solution of Ordinary Differential Equations (ODEs) that describe plant dynamics (1.1c) and optimizing the cost function (1.1a). Single shooting algorithms propose eliminating the states from decision variables by using numerical integrators to obtain states as functions of control inputs, which leads to a Nonlinear Programming (NLP) problem. Multiple shooting implies keeping the states as decision variables by dividing the prediction horizon into intervals and integrating for each integral independently, which leads to a larger NLP compared to single-shooting. However, the sparsity

pattern of the resulting problem can be exploited so that memory and computational requirements are significantly reduced. Direct transcription algorithms, instead of relying on external ODE solvers, approximate differential equations (1.1c) with algebraic equations directly, for example by using a polynomial approximation of the state trajectory. This leads to an even larger and sparser NLP: in addition to states and inputs, intermediate integration stages are also considered as decision variables.

For the Linear-Quadratic Regulation (LQR) problems, which can be considered as a special case of (1.1), the ODE (1.1c) can be solved exactly. In this case the NLP will be reduced to a Quadratic Programming (QP) problem. Analogously to single and multiple shooting, the QP problem can be formulated either by eliminating state variables or by considering states as decision variables. The former formulation is known as a *condensed formulation* and the latter is a *non-condensed formulation* [43].

Algorithms for solving QPs and NLPs arising from the optimal control applications can typically be classified into:

- First order algorithms
  - Fast-gradient algorithm [65].
  - Operator splitting algorithms [87].
- Second order algorithms
  - Interior-point algorithms [98].
  - Active set methods [67].
  - Sequential Quadratic Programming (SQP) [67], where QPs can be solved using interior-point or active set methods.

Choosing an optimization algorithm for a particular application depends on the problem (1.1) properties, temporal discretization method and available computational resources. It is important to exploit the coupling between optimal control problem formulation, optimization solver and underlying computing platform [48].

First order methods, being widely used for large-scale optimization, typically exhibit slow convergence, which is often compensated by the low complexity of an iteration. Recent studies

show that first order methods can also be successfully applied to small and medium scale control problems [87]. In contrast, second order methods have relatively complex iterations, which often involve second order derivatives evaluation and solving a system of linear equations. However, the number of iterations is significantly smaller compared to first order algorithms.

Several special cases of the optimal control problem (1.1) will be considered in this work and custom hardware implementations of the corresponding first and second order optimization solvers will be presented.

1.3 Outline of the thesis

Following the introduction, this thesis is organised as follows.

Chapter 2 reviews existing computing architectures with a focus on reconfigurable platforms. Languages for programming reconfigurable platforms are also briefly reviewed in this chapter.

Chapter 3 presents a toolchain for prototyping embedded optimization algorithms on reconfigurable and heterogeneous hardware.

Chapter 4 proposes and investigates a heterogeneous hardware implementation of a nonlinear MPC controller based on an interior point algorithm. Within the presented implementation a new way of exploiting the Karush-Kuhn-Tucker matrix structure on reconfigurable hardware is proposed.

Chapter 5 presents an application of bi-objective optimization to a predictive controller software and hardware co-design problem. The proposed approach allows automating the selection of design parameters both on software and hardware levels.

Chapter 6 summarizes the main contributions and discusses directions for future research.

1.4 Contributions of the thesis

The contributions of this research can be summarised as follows:

- A chain of software tools for quick prototyping of embedded optimization algorithms on reconfigurable and heterogeneous hardware. The toolchain consists of two pieces:
  - *SPLIT*, a code generation tool for optimization using operator splitting methods that is capable of generating code both for general-purpose processors and reconfigurable

platforms. The toolbox was developed in the Automatic Control Laboratory of the Swiss Federal Institute of Technology in Lausanne and the contributors are: Harsh Shukla, Colin Jones and Georgios Stathopoulos. The contribution of this thesis is enabling code generation for reconfigurable hardware and heterogeneous processors.

  - *Protoip*, a software tool for rapid prototyping of online optimization algorithms on reconfigurable hardware. Main contributions of this work include: (i) adding possibility of prototyping heterogeneous implementations; (ii) introducing a new communication layer to improve reliability of data transfer between embedded platform and desktop machine for processor-in-the-loop simulations; (iii) improving the way of managing application projects to make Protoip easily compatible with code generation tools, in particular with SPLIT.

- A framework for nonlinear predictive controller implementation on a heterogeneous computing platform. Most of the literature is focused on hardware acceleration of linear model predictive control, while this thesis proposes accelerating nonlinear MPC by splitting the computational workload between a fixed architecture CPU and a reconfigurable platform.

- Application of a systematic optimization-based approach to MPC software and computational hardware co-design problem. Design and implementation of a predictive controller is often associated with trading off performance against computational requirements. This thesis will propose a new way of exploring these design trade offs that allows reducing the workload of the human in the design loop.

1.5 List of publications

1.5.1 Conference papers

- **Khusainov, B.**; Kerrigan, E.C.; Suardi A.; Constantinides G.A., "Nonlinear Predictive Control on a Heterogeneous Computing Platform," *IFAC World Congress*, Toulouse, France, 2017. Corresponds to Chapter 4.

- Harsh, A.S.; **Khusainov, B.**; Kerrigan, E.C.; Jones C.N., "Software and Hardware Code Generation for Predictive Control Using Splitting Methods," *IFAC World Congress*, Toulouse, France, 2017. Corresponds to Chapter 3.

- **Khusainov, B.**; Kerrigan, E.C.; Constantinides G.A., "Multi-objective Co-design for Model Predictive Control with an FPGA," *European Control Conference*, Aalborg, Denmark, 2016. Corresponds to Chapter 5.

- Kerrigan, E.C.; Constantinides G.A.; Suardi A.; Picciau. A.; **Khusainov, B.**, "Computer Architectures to Close the Loop in Real-time Optimization," *IEEE Conference on Decision and Control*, Osaka, Japan, 2015. Corresponds to Chapter 2.

1.5.2 Other conference talks

- **Khusainov, B.**; Kerrigan, E.C.; Suardi A.; Constantinides G.A., "Nonlinear Model Predictive Control on a Heterogeneous Computing Platform," *SIAM Conference on Optimization*, Vancouver, Canada, 2017

- **Khusainov, B.**; Kerrigan, E.C.; Constantinides G.A., "Exploring Trade-offs in Predictive Control on Reconfigurable Computing Platforms," *European Conference on Computational Optimization*, Leuven, Belgium, 2016

- Suardi A.; **Khusainov, B.**; Kerrigan, E.C.; Constantinides G.A., "Fast prototyping of embedded optimization algorithms on reconfigurable and heterogeneous computing platforms," *European Conference on Computational Optimization*, Leuven, Belgium, 2016

- **Khusainov, B.**; Kerrigan, E.C.; Constantinides G.A., "Multi-Objective Co-Design for Embedded Optimization-based Control," *International Conference on Continuous Optimization*, Tokyo, Japan, 2016

Chapter 2

Computer Architectures for Online Optimization and Corresponding Programming Tools

This chapter aims to:

- Review existing computing architectures with a focus on reconfigurable platforms and discuss the applicability of different architectures for online optimization.
- Review existing methods and tools for programming reconfigurable platforms.

2.1 Computing platforms

2.1.1 General-purpose Central Processing Units (CPUs)

The CPU is the most widely used computing platform for embedded optimization. The key components of most CPUs are (Figure 2.1):

- Arithmetic Logic Unit (ALU), which performs arithmetic and logic operations on integer binary numbers.
- Control Unit (CU), which extracts instructions from memory, decodes them in order to direct operation of ALU and memory.
- Cache memory, which is connected to external Random Access Memory (RAM).

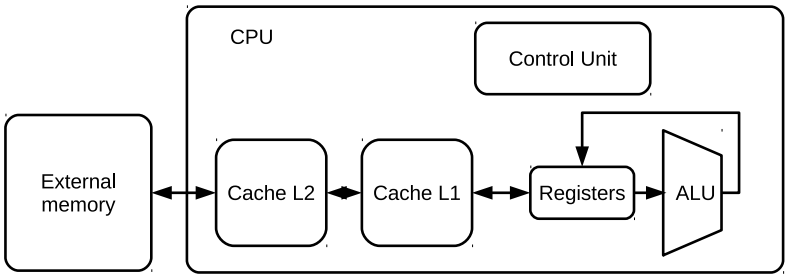


Fig. 2.1 CPU architecture.

Since the early 1980s, the performance gap between external memory and CPUs has been increasing: new processors have been getting significantly faster, while memory has been evolving at a much slower rate [73]. This inspired the development of cache memory hierarchy, which aims to bridge the performance gap between external memory and the CPU. This hierarchy represents a chain of memory blocks that can be described by a trade-off between data read/write time and capacity, where the fastest memory is placed close to ALU registers. Efficient utilization of cache memory hierarchy depends on an algorithm memory access pattern, implementation source code and compilation tools.

Most general purpose processors use 64-bit double precision floating point arithmetic for number representation, which is often not optimal in terms of accuracy vs power/silicon usage trade-offs. A binary representation of a floating point number f consists of three components: sign bit (s), mantissa (m) and exponent (e) leading to the following representation:

$$f = s \times 2^{\text{bias}-e} \times 1.m.$$

The first bit of the mantissa is assumed to be 1 and not explicitly stored in the *normalized form*. This representation allows for a high *dynamic range*, i.e. the ratio of the maximal to minimal representable numbers. However, dynamic range comes at the price of increased computation time, silicon usage and power consumption: since the exponents of two numbers in general are not the same, every addition operation is preceded by denormalization and followed by normalization.

Historically CPUs were developing under assumption of sequential programming, so most of the speedup was achieved by improving transistor size and timing characteristics, which allowed packing more transistors in a chip and increasing clock frequency. This was happening in accordance with Moore's law, which states that the number of transistors in an integrated circuit doubles every

two years [64]. However, in recent years clock frequency stopped scaling, which motivated development of multi-core processors, where several cores often share a common cache memory within the same chip. Although Moore's law is still valid for multi-core systems, achieving theoretical peak performance is not straightforward due to the following reasons:

- An algorithm can be inherently unparallelisable.
- Communication overhead might cancel out parallelism speedup.
- Source code compilation tools might be unable to capture algorithm parallelism and compile inefficient implementations.

Taking the above into account, general-purpose CPUs, currently being the most common platform for MPC, could be potentially inefficient for acceleration of future optimization-based controllers due to fixed memory architecture, predefined number representation and limited parallelism.

Examples of CPU-based MPC implementations presented in the literature:

- [37] presents predictive vibration control implementations on ARM Cortex-M processors. Since only limited processing power is available on Cortex-M processors, "simple" constrained LQR formulation is considered in this work.
- Nonlinear predictive control of a gantry crane is presented in [28]. Intel Xeon 2.53 GHz quad core processor was employed in order to solve nonlinear optimal control problems numerically.

The above examples highlight computational resource vs control performance trade-offs: the former paper considers classical linear predictive control on an embedded CPU, while the latter employs a multi-core desktop machine in order to capture system dynamics with nonlinear MPC.

2.1.2 Microcontrollers

A microcontroller is a single chip that, in addition to a central processing unit, incorporates peripherals, which might include memory, analogue to digital converters (ADCs) and other input/output subsystems. This allows direct connection of sensors and actuators, which is desirable for industrial applications. Microcontrollers have very limited processing power and data storage space compared

to desktop CPUs, which restricts the application scope to relatively simple algorithms. On the other hand, low power consumption makes microcontrollers particularly useful for autonomous embedded applications. For example, the explicit MPC algorithm has been implemented for vibration control using an STM32 microcontroller [90].

2.1.3 Graphical Processing Units (GPUs)

A graphical processing unit is an array of special multi-threaded processors, each of which can perform the same operation on multiple data entries simultaneously. Communication between processors is performed via shared global memory. Although this architecture was initially tailored for image processing applications, modern general-purpose GPUs (GPGPUs) allow the implementation of other types of algorithms. For example, the Nvidia Tesla GK210 GPU incorporates thirteen multithreaded processors, each of which can process 2048 instructions simultaneously [69]. This makes GPGPU's peak performance much higher even compared to the fastest multi-core CPUs. However, achieving peak performance requires a regular memory access pattern, which is the case for very few algorithms; the classical example would be dense matrix-vector multiplication. Irregular memory accesses and conditional branches often lead to significant performance decrease. Another limitation of GPUs, especially in relation to embedded applications, is the necessity of having a separate host CPU that manges the GPU(s).

2.1.4 Programmable Logic Controllers (PLCs)

Programmable logic controllers are self-contained industrial devices intended for automatic process control. Depending on the computational requirements, a microcontroller or CPU can be used as a processing core inside a PLC. However, PLCs have a set of distinguishing features compared to microcontrollers/CPUs. Firstly, PLCs are programmed using graphical ladder logic languages, which resemble relay systems that were used before PLCs came into use. Secondly, PLCs are manufactured as complete devices, which can be used in industry directly, whereas a microcontroller represents a single chip coming as a part of electronic circuit. Finally, PLCs can operate with high DC/AC voltages, while typical microcontrollers are compatible only with TTL and/or CMOS technologies [88, 5].

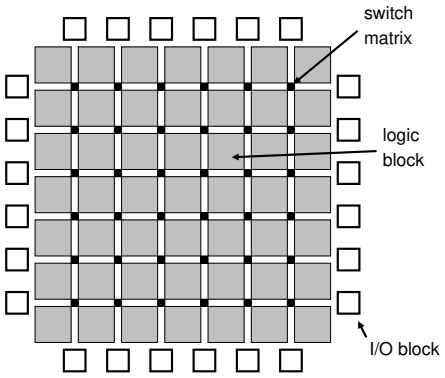


Fig. 2.2 FPGA architecture.

2.1.5 Field-Programmable Gate Arrays (FPGAs)

An FPGA is a two-dimensional array of Configurable Logic Blocks (CLBs), connected via flexible switches (Figure 2.2). CLBs are built from Lookup Tables (LUTs) and Flip-Flops (FFs); for example, the Xilinx Artix-7 FPGA [102] programmable logic CLBs consist of four LUTs and eight FFs. A LUT is a flexible logic unit that can be employed as a logic function, shift register or RAM/ROM memory. Sequential logic is implemented using FFs, which can be cascaded to build registers. Registers are mainly used for storing data between consecutive clock cycles. Although in theory any digital circuit can be implemented by means of CLBs, using special purpose components can be more reasonable from a resource usage, circuit routing and chip clocking point of view. For instance, the Artix-7 FPGA Block RAMs (BRAMs) are intended for dense data storage — each block has two ports and can store up to 36Kb of data [23]. Depending on the application, a BRAM can be configured as two decoupled memory blocks of 18Kb each. An equivalently sized LUT memory would occupy a relatively large area and suffer from timing issues. Another example is an arithmetic block: although long wordlength arithmetic operations also can be implemented by means of standard blocks, using a DSP unit with multiplication and post/pre-addition capabilities results in a smaller area and higher clock frequency [23, 102].

The key outstanding feature of FPGAs is customizability. This applies to the data processing units, memory sub-system and number representation. FPGAs allow implementation only of the computational units required by a specific algorithm, unlike CPUs that have fixed logic for performing a predefined set of operations. Another important benefit of using FPGAs is memory flexibility. Data can be partitioned into separate memory blocks and placed near the corresponding processing units. This provides a huge potential for parallelization of computations. Furthermore,

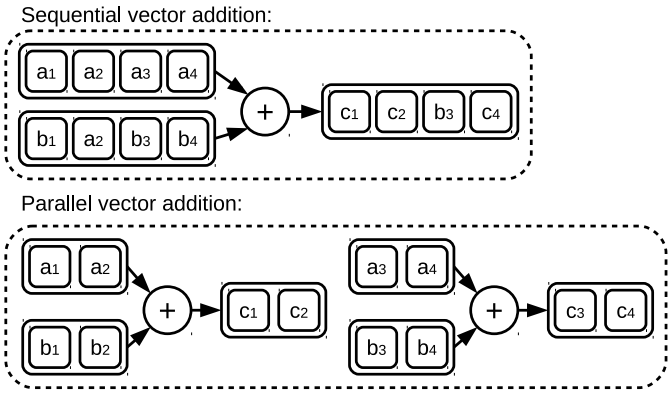


Fig. 2.3 Data-level parallelism for vector-vector addition.

since both memory word size and computational units are flexible, number representation also becomes variable. However, extended flexibility comes at the price of a reduced clock rate compared to CPU-like architectures. Overcoming this limitation requires achieving a sufficient degrees of parallelisation, by introducing data- and/or task-level parallelism [22]. These acceleration techniques will be discussed next.

Data-level parallel processing

Data-level parallelism refers to splitting computations across multiple processors, so that different sets of data are processed simultaneously. This technique can also be implemented with multi-core CPUs and GPUs. Consider the example of parallelising vector-vector addition in Figure 2.3. Assuming that reading data and performing addition takes one clock cycle in total and writing data takes zero clock cycles, sequential implementation would require four clock cycles. Replicating the adder results in a faster implementation that would take only two clock cycles. Further speedup can be achieved by increasing the number of adders up to four. For this toy example, where the algorithm does not have data dependencies, data-level parallelism might work well. However, for real-world algorithms the potential speedup is limited by the part of the algorithm that cannot benefit from parallelization (Amdahl's law [2]). Provided that the parallelizable fraction of a certain algorithm is given by P and the number of parallel processors is given by R, acceleration in comparison to fully sequential implementation would be given by:

$$\frac{1}{(1-P)+\frac{P}{R}}$$

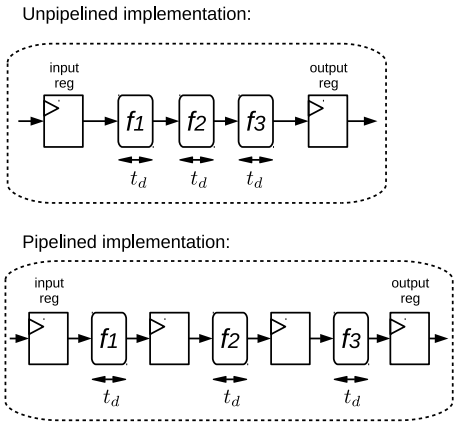


Fig. 2.4 Unpipelined and pipelined implementations of an abstract algorithm that consists the sequential operations: f_1, f_2 and f_3.

For instance, if only half of a certain algorithm (in terms of the computation time) is parallelizable the best possible acceleration would be 2x.

It should also be noted that parallelization, in addition to replicating the processing units, often requires memory partitioning (Figure 2.3), which could be crucial for certain applications, since $P > 1$ memory blocks of a certain total capacity would require more silicon resources compared to one large memory block of the same capacity. This becomes even more important, taking into account fixed size BRAM blocks of modern FPGAs.

Data pipelining

In a digital synchronous circuit the changes in the states of registers (flip-flops) are synchronized by a clock signal. The output of a certain register remains constant until the clock signal changes its state, when the output update happens. Hence, the clock period has to be longer than the propagation delay between consecutive registers. The overall computation time, known as *latency*, is given by:

$$\text{clock period} \times \text{number of clock cycles}.$$

Consider two hardware implementations of an abstract algorithm that involves consecutively applying three operators f_1, f_2 and f_3 to the input data (Figure 2.4). Propagation delays for all operators are the same and equal to t_d seconds. Provided that the input register already holds relevant data it will take one clock cycle for the result to appear in the output register of the upper

circuit. Assuming that the circuit clock period is equal to $3t_d$, which is the smallest possible period, the overall *latency* is given by:

$$3t_d \times 1 \text{ clock cycle} = 3t_d$$

In the lower circuit propagation delay between consecutive registers is shortened by inserting intermediate registers between operators. This allows reducing the clock period to t_d, while increasing the number of required clock cycles to three, which leads to the same latency as the upper circuit:

$$t_d \times 3 \text{ clock cycles} = 3t_d$$

However, the lower circuit can accept a new input every t_d seconds, while the upper can start processing new piece of data only every $3t_d$ seconds. This becomes crucial when processing large amounts of data. Processing 1000 inputs would take $3000t_d$ seconds for the upper circuit, but $1002t_d$ seconds for the lower. The technique implemented in the lower circuit is known as *pipelining*. Efficiency of pipelining is measured by *throughput*, the rate at which a circuit can accept new inputs. The *Initiation Interval (II)* is the time between accepting consecutive inputs. Throughput and initiation interval have the following relation:

$$\text{throughput} = \frac{1}{\text{initiation interval} \times t_d}$$

Note that the initiation interval is measured in clock cycles, while throughput is measured in s^{-1}.

Algorithm data dependencies might affect the efficiency of pipelining and hence the execution time of the entire algorithm. Chapter 4 presents one possible way of addressing this issue, which is based on offline scheduling.

Number representation

Since reconfigurable platforms allow flexibility of processing units and memory subsystem, data representation also becomes a flexible design parameter. As was mentioned above, the vast majority of today's computational platforms use floating data representation, where numbers are stored in three separate fields: sign, mantissa and exponent. Reconfigurable platforms support varying mantissa and exponent size to tailor precision for a particular application. Further resource savings

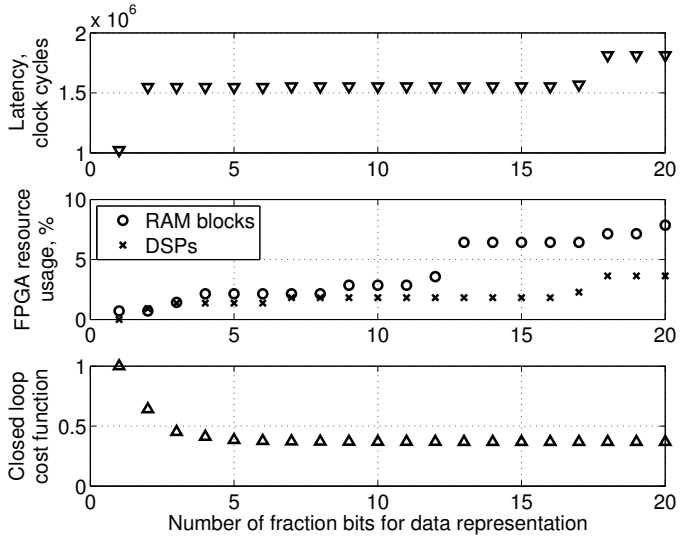


Fig. 2.5 Latency (execution time) of a fast gradient optimization-based controller, FPGA resource usage and closed-loop system performance as a function of number of fraction bits. The number of integer bits is fixed to 8. A lower closed-loop cost implies better performance.

can be achieved by fixing exponents and operating with mantissas directly, which leads to *fixed point* data representation. Since the exponents become fixed, there is no need to denormalize and normalize numbers before and after every addition/subtraction operation (refer to Section 2.1.1 for details on this). This makes the complexity of fixed point operations identical to that of integer numbers. On the other hand, fixed-point arithmetic is highly affected by *overflow* and *round off* errors. Overflow errors happen due to lack of dynamic range and can be avoided by pre-calculating the largest possible absolute value and allocating an appropriate scaling factor. It is also essential to mathematically prove and/or practically verify stability of iterative optimization algorithms under the presence of round-off errors. Even transformation of the original problem to fixed point representation may lead to the loss of favorable properties of an optimization problem. For instance, the objective function might lose convexity or the feasible set might be distorted [45].

Figure 2.5 demonstrates how the number of fraction bits for a fixed point data representation affects algorithm execution time, resource utilization and performance of a closed-loop control system. The presented graphs were obtained from simulations of a fast gradient-based model predictive controller [45] for a mass-spring-damper system with 10 states and 5 inputs. More details on the experimental setup can be found in [47] and Section 5. The notable point on the graph is that after reaching a certain number of bits (6 bits in this case), no significant performance improvement is detected. Hence, there is no need to waste resources by introducing redundant precision. This

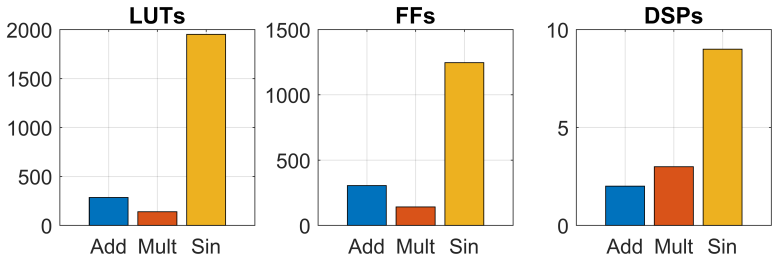


Fig. 2.6 FPGA resource usage for different operations. Data is obtained using vendors synthesis tool, Vivado HLS.

example illustrates that number representation for real-time solvers should be chosen with respect to the closed-loop system response, rather than just considering the accuracy of the optimization solver on its own. Chapter 5 of the this thesis presents an approach for systematic selection of data representation as well as other design parameters for predictive control.

Depending on the application, it might be justifiable to use other types of data representation than fixed and floating point. For instance, a logarithmic data representation is well suited for multiplication and division operations, while preserving high dynamic range [39]. Another example of alternative arithmetic is dual-fixed point, which combines the advantages of fixed and floating point computations [29].

2.1.6 Heterogeneous processors

When designing an FPGA circuit it is important to take into account resource usage of different computation and memory blocks. For instance, consider resource usage estimations for three different operations: addition, multiplication and sine (Figure 2.6). Estimates are obtained for a single precision floating point number representation using Vivado HLS synthesis tool with the Xilinx Artix-7 as a target platform. The sine operation, which is widely used in robotics applications, consumes far more resources in terms of LUTs, FFs and DSP units compared to addition and multiplication. For this reason, it could be reasonable to distribute the computational workload between a conventional CPU (for resource hungry operations) and an FPGA (for resource efficient operations). Different types of platforms can be implemented within a single *System-on-a-Chip (SoC)*. Then main advantages of SoCs or *heterogeneous processors* over using physically separate devices are: lower cost, faster data transfer between subsystems, better reliability and lower power consumption [23].

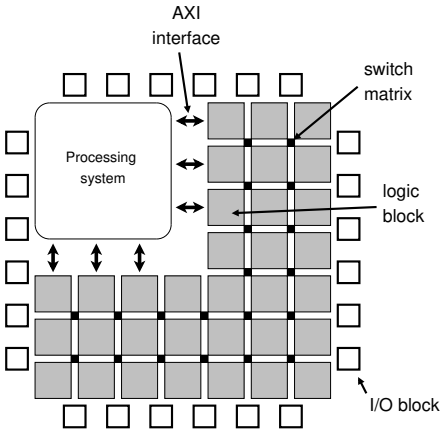


Fig. 2.7 Zynq SoC architecture.

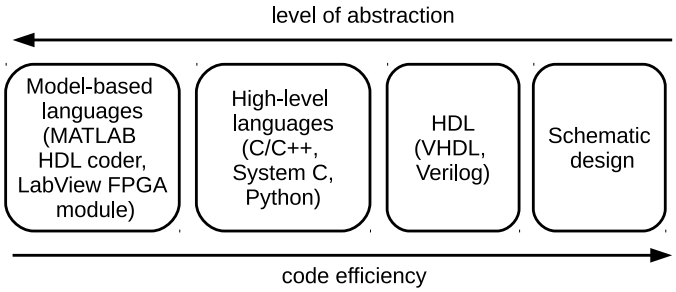


Fig. 2.8 FPGA programming languages.

Consider the Zynq platform from Xilinx, an example of an SoC (Figure 2.7). It incorporates a dual-core ARM processor, which is a part of the *processing system* (PS), and an Artix-7 FPGA *programmable logic* (PL). Data exchange between the processing system and FPGA logic is implemented using an Advanced eXtensible Interface (AXI) with burst mode that allows reading/writing of data every clock cycle if the memory access pattern is regular. Chapter 4 presents an implementation of nonlinear MPC on this platform.

2.2 Programming languages and tools for FPGAs and heterogeneous computers

FPGAs can be programmed using different languages, trading off circuit efficiency against design time (Figure 2.8). Schematic design requires describing circuit topology by manually connecting the wires that carry voltages between circuit components, this cannot be considered as a scalable approach, especially taking into account modern FPGA sizes. Hardware description languages (HDLs) abstract low-level implementation details, allowing users to focus on the circuit behaviour.

In contrast to software programming languages that define a set of operations to be performed by a processor (control flow), HDLs describe the architecture of the processor itself (data flow). VHDL and Verilog are the most widely used HDLs. High-level languages (e.g. C, Python) raise the level of abstraction even further, which makes circuit design productivity comparable to that of software implementations. However, time and resource efficiency of the resulting circuits becomes highly dependent on source code quality and additional synthesiser directives that are defined by a user [76]. The case study in [99] compares efficiencies of two floating point accumulators synthesised from software-oriented and hardware-oriented pieces of C-code respectively. Synthesis results show that hardware oriented code is more than 3.5 times faster, while achieving the same resource scaling. Finally, model-based languages are on the top of the hierarchy; they allow synthesis of circuits based on high level graphical or code-based models. Moreover, they often can handle heterogeneous processors by automatically splitting the workload between different subsystems. Unfortunately, quality of the resulting circuits is not comparable to any of the lower level languages.

In this thesis we use a C-based approach, both for programming FPGAs and CPUs. This can be beneficial because of the following reasons:

- Existing C libraries and code generation tools can potentially be employed for FPGA design. This will be discussed in detail in Chapter 3.
- When targeting heterogeneous platforms, C code can be shifted between software (CPU) and hardware (FPGA) with minor changes.
- A relatively high level of abstraction allows the user to focus on circuit functionality rather than implementation details.
- Provided that the C code is written with respect to the FPGA architecture with appropriate synthesis directives, performance similar to an HDL approach can be achieved.

The FPGA design flow involves several consecutive stages:

1. *High-level synthesis*: Converting the C code with synthesis directives (e.g. loop unrolling or pipelining) into low level HDL code.
2. *Synthesis*: The process of transforming HDL code into a *netlist*, a graph that defines the connection of all logic gates and registers.

3. *Place-and-route* (P&R): Solving a set of optimization problems in order to fit the netlist to a particular FPGA platform. The outcome of P&R is a bitstream that can be uploaded onto the FPGA.

4. *Functional testing* of the circuit. For optimization-based control applications, the FPGA circuit has to be tested in the loop with a plant model.

Complexity of the design flow is one of the factors that limits wider adoption of reconfigurable platforms for optimal control. The next chapter will present a tool for automation of the process of synthesizing, deploying and verifying of optimization algorithms on an FPGA.

2.3 Summary

This chapter reviewed existing computing architectures and discussed their suitability for online optimization. FPGAs, in contrast to fixed-architecture platforms, provide users with the flexibility of tailoring hardware for algorithms in order to optimize resource usage vs computation time trade-offs. However, for certain computations (e.g. nonlinear operators) synthesizing FPGA circuits might be relatively resource consuming, which justifies distributing the computational workload between a conventional CPU and an FPGA by using heterogeneous computers.

Being a target hardware platform for online optimization, FPGAs can also be used as a *testbed*. Testing a set of non-reconfigurable platforms to find the most suitable for a certain algorithm is not a reasonable approach, both from a time and expense points of view. FPGAs allow estimation of the impact of design variables on the hardware platform performance and silicon usage using one single chip. This significantly reduces time-to-market of non-reconfigurable platforms. An example of using an FPGA for hardware parameters tuning will be presented in Chapter 5.

Chapter 3

Hardware Code Generation for Embedded Optimization with Operator Splitting Methods

This chapter presents a toolchain for quick prototyping of embedded optimization algorithms on reconfigurable and heterogeneous computing platforms. The toolchain consists of two parts:

- *SPLIT* [85, 87], a code generation tool for embedded optimization using operator splitting methods. SPLIT supports code generation for CPUs, FPGAs and heterogeneous platforms. The toolbox was developed in the Automatic Control Laboratory of the Swiss Federal Institute of Technology in Lausanne and the contributors are: Harsh Shukla, Colin Jones and Georgios Stathopoulos. This thesis is focused on enabling code generation for FPGAs and heterogeneous platforms.
- *Protoip* [51], a software tool for rapid prototyping of online optimization algorithms on FPGAs and heterogeneous computers. Protoip abstracts many low-level implementation details and allows automating algorithm compilation, deployment and testing.

A high-level description of the design flow with the proposed toolchain is presented in Figure 3.1.

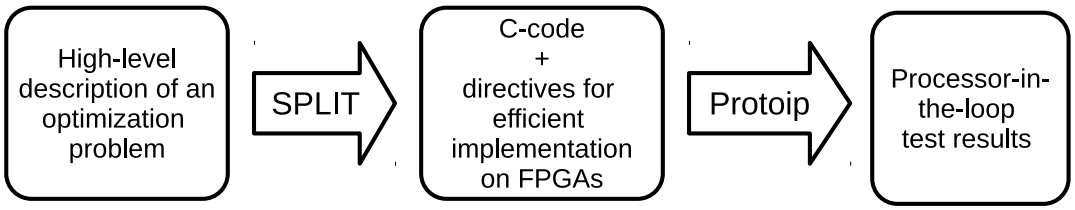


Fig. 3.1 Toolchain design flow.

The chapter is organized as follows. Section 3.1 describes the main features of SPLIT, focusing on algorithmic and implementation aspects. A new release of Protoip is introduced in Section 3.2. Following this, Section 3.3 presents experimental results and Section 3.4 concludes the chapter.

3.1 SPLIT code generator

3.1.1 Introduction

Operator splitting methods has traditionally been used for large scale optimization, e.g. [72], which is mainly due to the fact that splitting methods do not use computationally expensive second order sensitivity information, relying only on first order derivatives. However, in recent years, splitting algorithms have been gaining attention in relation to small and medium scale optimization problems and numerical optimal control, in particular [70].

The key feature of splitting algorithms is decomposing the original optimization problem into simpler computationally inexpensive subproblems. This facilitates implementing splitting algorithms on low-cost embedded platforms. There are several C code generation tools for splitting methods presented in the literature, e.g. QPGEN [34], FiOrdOs [93] and OSQP [12]. These tools generate code for CPU-based implementations. The SPLIT toolbox generates C-code for several different types of embedded computers (including CPUs and FPGAs) based on a high-level description of optimization problems in Matlab environment. Several operator splitting methods, including the widely-used alternating direction method of multipliers [18], are supported. Using Protoip toolbox on the underlying level smoothens the transition from C code to deployment on FPGAs and performing processor-in-the-loop tests.

3.1.2 Embedded optimization on FPGAs: existing work

There are several FPGA implementations of embedded optimization algorithms presented in the literature. Most of these algorithms consider linear MPC as the main application:

- [96] presents an implementation of linear MPC on a system-on-a-chip that consists of a CPU and an FPGA. Eliminating the states from the decision variables and incorporating inequality constraints in the cost function leads to an unconstrained nonlinear optimization problem, which is solved using Newton's method. For the Newton's method the authors propose evaluating derivatives on the CPU and accelerating solving linear systems on the FPGA.

- [44] describe a two-stage architecture for interior point algorithm-based predictive control. Interior point algorithms have a similar computational pattern with the Newton's method. However, the considered architecture is tailored for solving quadratic programming problems (in contrast to general nonlinear programming problems), which simplifies evaluation of the derivatives. The linear system solver is based on an iterative algorithm, hence the computational logic is reused efficiently.

- [45, 75] propose several implementations of first order methods-based MPC on FPGAs. The main computational kernel for this class of algorithms is matrix-vector multiplication, which has a huge parallelization potential. [45] and [75] differ in the way how this operation is implemented: the former implementation is based on an adder tree, while the latter relies on multiply-accumulate units.

SPLIT differs from the above implementations in the following ways:

- Hardware code is generated based on a high-level description of an optimization problem, which makes the tool accessible to engineers with various backgrounds.

- SPLIT is not tailored for a particular sparsity pattern. Instead, the sparsity pattern is identified in the code generation stage and corresponding circuit description code is generated.

3.1.3 Problem formulation

SPLIT deals with optimization problems of the form:

$$\text{minimize} \quad f(\theta)+g(\eta) \tag{3.1a}$$

$$\text{subject to} \quad L\theta-\eta=-l \tag{3.1b}$$

with $\theta \in \mathbb{R}^{n_\theta}$, $\eta \in \mathbb{R}^{n_\eta}$, $l \in \mathbb{R}^{n_\eta}$ and $L \in \mathbb{R}^{n_\eta} \times \mathbb{R}^{n_\theta}$. The objective is a sum of two functions: $f : \mathbb{R}^{n_\theta} \to (-\infty;\infty]$ and $g : \mathbb{R}^{n_\eta} \to (-\infty;\infty]$, which are proper, lower semi-continuous convex functions.

The Lagrangian associated with (3.1) is given by:

$$\mathcal{L}(\theta,\eta,\lambda) = f(\theta)+g(\eta)+\lambda^T(L\theta-\eta+l), \tag{3.2}$$

where $\lambda \in \mathbb{R}^{n_\eta}$ is a vector of Lagrange multipliers.

We also introduce the *augmented* Lagrangian:

$$\mathcal{L}_a(\theta,\eta,\lambda) = f(\theta)+g(\eta)+\lambda^T(L\theta-\eta+l)+\frac{\rho}{2}||L\theta-\eta+l||_2^2, \tag{3.3}$$

with parameter ρ.

The dual function of (3.1) can be derived from the Lagrangian (3.2) as

$$d(\lambda) = \min_{\theta,\eta}\{f(\theta)+g(\eta)+\lambda^T(L\theta-\eta+l)\}. \tag{3.4}$$

Exploiting the decoupled nature of (3.1), the dual function (3.4) can be reformulated as

$$d(\lambda) = \min_{\theta}\{f(\theta)+\lambda^T L\theta\}+\min_{\eta}\{g(\eta)-\lambda^T\eta\}+\lambda^T l. \tag{3.5}$$

In addition to primal and dual formulations, *saddle point* formulation can be obtained by means of *partial* dualization:

$$\mathcal{S}(\theta,\lambda) = f(\theta)+\lambda^T(L\theta+l)-\min_{\eta}\{g(\eta)-\lambda^T\eta\}. \tag{3.6}$$

Formulation (3.1) can capture a large range of convex optimization problems including the constrained linear quadratic regulation, which will be used in this work for benchmarking:

$$\underset{u_0 \dots u_{N-1}, x_0 \dots x_N}{\text{minimize}} \quad \sum_{k=0}^{N-1} \left(\frac{1}{2} x_k^T Q_d x_k + \frac{1}{2} u_k^T R_d u_k \right) + \frac{1}{2} x_N^T P_d x_N \tag{3.7a}$$

$$\text{subject to} \quad x_0 = \hat{x}, \tag{3.7b}$$

$$x_{k+1} = A_d x_k + B_d u_k, \quad \text{for } k = 0, 1, \dots, N-1, \tag{3.7c}$$

$$x_k \in X, \quad \text{for } k = 0, 1, \dots, N, \tag{3.7d}$$

$$u_k \in U, \quad \text{for } k = 0, 1, \dots, N-1 \tag{3.7e}$$

where $x_k \in \mathbb{R}^n$, $u_k \in \mathbb{R}^m$, $A_d \in \mathbb{R}^{n \times n}$, $B_d \in \mathbb{R}^{n \times m}$. Q_d, R_d, W_d and P_d are the weight matrices of appropriate dimensions that have to be chosen to ensure convexity of the objective function. The sets X and U are convex and define constraints on the states and inputs, respectively. $\hat{x}$ is an initial state and N is a prediction horizon.

In order to match the formulation (3.1), the problem (3.7) can be rewritten in the *consensus form* [18]:

$$\underset{\substack{u_0 \dots u_{N-1}, x_0 \dots x_N \\ \bar{u}_0 \dots \bar{u}_{N-1}, \bar{x}_0 \dots \bar{x}_N}}{\text{minimize}} \quad \sum_{k=0}^{N-1} \left(\frac{1}{2} x_k^T Q_d x_k + \frac{1}{2} u_k^T R_d u_k \right) + \frac{1}{2} x_N^T P_d x_N \tag{3.8a}$$

$$\text{subject to} \quad x_0 = \hat{x}, \tag{3.8b}$$

$$x_{k+1} = A_d x_k + B_d u_k, \quad \text{for } k = 0, 1, \dots, N-1, \tag{3.8c}$$

$$x_k = \bar{x}_k, \quad \text{for } k = 0, 1, \dots, N, \tag{3.8d}$$

$$u_k = \bar{u}_k, \quad \text{for } k = 0, 1, \dots, N-1, \tag{3.8e}$$

$$\bar{x}_k \in X, \quad \text{for } k = 0, 1, \dots, N, \tag{3.8f}$$

$$\bar{u}_k \in U, \quad \text{for } k = 0, 1, \dots, N-1 \tag{3.8g}$$

With the above reformulation: $\theta \coloneqq (u_0 \dots u_{N-1}, x_0 \dots x_N)^T$ and $\eta \coloneqq (\bar{u}_0 \dots \bar{u}_{N-1}, \bar{x}_0 \dots \bar{x}_N)^T$, the plant dynamics and the control objective can be captured using the function f:

$$f(\theta) \coloneqq h(\theta) + \delta(\theta) \tag{3.9a}$$

$$h(\theta) = \sum_{k=0}^{N-1} \left(\frac{1}{2} x_k^T Q_d x_k + \frac{1}{2} u_k^T R_d u_k \right) + \frac{1}{2} x_N^T P_d x_N \tag{3.9b}$$

$$\delta(\theta) = \begin{cases} 0 & x_0 = \hat{x}, \quad x_{k+1} = A_d x_k + B_d u_k, \quad \text{for } k = 0, 1, \dots, N-1 \\ \infty & \text{otherwise} \end{cases} \tag{3.9c}$$

Constraints (3.7d) and (3.7e) can be incorporated into the indicator function g:

$$g(\eta) = \begin{cases} 0 & \bar{x}_k \in X \text{ for } k = 0, 1, \dots, N, \quad \bar{u}_k \in U \text{ for } k = 0, 1, \dots, N-1 \\ \infty & \text{otherwise} \end{cases} \tag{3.10}$$

Constraints (3.8d) and (3.8e) correspond to (3.1b).

The general problem (3.1) can be solved using splitting methods, which take advantage of the fact that the objective is a sum of two separable functions. Three splitting algorithms are considered in this work: Alternating Minimization Algorithm (AMA), Alternating Direction Method of Multipliers (ADMM) and the Primal-Dual Algorithm (PDA), which is mainly known as the Vũ-Condat algorithm [87].

3.1.4 Splitting algorithms

Alternating Minimization Algorithm (AMA)

The alternating minimization algorithm [92, 87] implies iterating over primal and dual variables and consists of three backbone operations (Algorithm 1). In the first two steps, minimization of the Lagrangian and augmented Lagrangian with respect to primal variables is performed. Following this, the third step couples primal variables by performing a *consensus step*, i.e. updating Lagrange multipliers.

For the problem (3.8), which is a special case of (3.1), all the steps above often have a closed form solution. From the computational point of view, the first step represents solving a system of linear equations. In contrast to the most of the second order methods, the linear system matrix for AMA is not changing with iterations. This opens the possibility of shifting the computational

Algorithm 1 Alternating minimization algorithm (AMA).

1: Require: λ^0, η^0, ρ. Refer to [87] for details on initial guess and parameter selection.
2: **for** $k = 0$ to N_{AMA} **do**
3: $\quad \theta_{k+1} = \underset{\theta}{\operatorname{argmin}} \quad \mathcal{L}(\theta, \eta^k, \lambda^k)$
4: $\quad \eta_{k+1} = \underset{\eta}{\operatorname{argmin}} \quad \mathcal{L}_a(\theta_{k+1}, \eta, \lambda^k)$
5: $\quad \lambda_{k+1} = \lambda_k + \rho(L\theta^{k+1} - \eta^{k+1} + l)$
6: **end for**

Algorithm 2 Alternating direction method of multipliers (ADMM)

1: Require: λ^0, η^0, ρ. Refer to [87] for details on initial guess and parameter selection.
2: **for** $k = 0$ to N_{ADMM} **do**
3: $\quad \theta_{k+1} = \underset{\theta}{\operatorname{argmin}} \quad \mathcal{L}_a(\theta, \eta^k, \lambda^k)$
4: $\quad \eta_{k+1} = \underset{\eta}{\operatorname{argmin}} \quad \mathcal{L}_a(\theta_{k+1}, \eta, \lambda^k)$
5: $\quad \lambda_{k+1} = \lambda_k + \rho(L\theta^{k+1} - \eta^{k+1} + l)$
6: **end for**

workload to the offline stage by pre-inverting or pre-factorizing the matrix. The complexity of the second step depends on the shape of the feasible region defined by (3.8f) and (3.8g). The crucial operation is projection on the feasible set, which is known to be well-mapped on reconfigurable platforms for several types of constraints including box and cone constraints [45]. The third step can potentially involve matrix-vector multiplication. However, due to the structure of (3.8d) and (3.8e), the matrix L from line 5 is sparse and often can be narrowed down to the identity, which does not require any matrix manipulations. As a result, the third step can be implemented with element-wise vector-vector manipulations and hence can be pipelined/parallelized efficiently.

Alternating direction method of multipliers (ADMM)

Although the theoretical backgrounds behind AMA and ADMM are different, these two algorithms have similar computational patterns. The difference comes from the first step (Algorithm 2): while AMA proposes minimizing the Lagrangian, ADMM deals with the augmented Lagrangian [35, 87]. This affects both the numerical behaviour and, potentially, the computational complexity. Introducing a regularization term to the Lagrangian makes ADMM capable of dealing with convex $f(\cdot)$ and $g(\cdot)$, while AMA requires strong convexity for the objective functions. On the other hand, the quadratic regularization term in the Lagrangian (3.3) might affect sparsity of the resulting linear system at the first step and hence increase computational complexity of ADMM iteration.

Algorithm 3 Prima-Dual Algorithm (PDA)

1: Require: $\lambda^0, \theta^0, \tau, \rho$. Refer to [87] for details on initial guess and parameter selection.
2: **for** $k = 0$ to N_{PDA} **do**
3: $\quad \theta^{k+1} = \mathbf{prox}_{\tau\delta}\Big(\theta^k - \tau(\nabla h(\theta^k) + L^T\lambda^k)\Big)$
4: $\quad \lambda^{k+1} = \mathbf{prox}_{\rho g^\star}\Big(\lambda^k + \rho(L(2\theta^{k+1} - \theta^k) + l)\Big)$
5: **end for**

Primal-Dual Algorithm (PDA)

Both AMA and ADMM rely on solving systems of linear equations at the first step, which might be undesirable for computers with a limited memory and/or processing power, especially if the Hessian of f is dense. In contrast, PDA relies only on the gradient information without explicit or implicit inversion of the matrices that involve the objective Hessians [20, 87]. We introduce two auxiliary definitions before describing PDA.

Definition 3.1.1. *Proximal operator. For a given function $f : \mathbb{R}^n \to \mathbb{R}$ and parameter ρ a proximal operator $\boldsymbol{prox}_{\rho f} : \mathbb{R}^n \to \mathbb{R}^n$ is defined as:*

$$\mathbf{prox}_{\rho f}(x) \coloneqq \underset{z}{\operatorname{argmin}}\{f(z) + \frac{\rho}{2}||z-x||_2^2\}. \tag{3.11}$$

Definition 3.1.2. *Conjugate function. For a given function $f : \mathbb{R}^n \to \mathbb{R}$ a conjugate function $f^\star : \mathbb{R}^n \to \mathbb{R}$ is defined as:*

$$f^\star(\lambda) \coloneqq \sup_z\{z^T\lambda - f(z)\}. \tag{3.12}$$

Unlike the splitting algorithms described above, PDA exploits the fact that f is a sum of smooth and non-smooth functions, as shown in (3.9). At the fist stage, a gradient direction step with respect to the smooth function is performed followed by projection on the dynamics subspace, which is described by a non-smooth indicator function (Algorithm 3). The dual variable is updated at the second step, using a proximal operator, which is based on g^*, a conjugate to the indicator function g (3.10). Considering the algorithm from an implementation point of view, the first step involves two matrix vector multiplications, while the second step is based on vector-vector manipulations, which includes projecting on g^*, assuming constraints on states and inputs are 'simple', e.g. box or conic constraints. The above discussion also assumes that L is the identity; for a more general case, which is also handled by SPLIT, each of the steps will have additional, potentially sparse, matrix-vector multiplication.

3.1.5 FPGA code generation

Generating C code for hardware implementations has the following particularities:

- Certain features of the C language may be not supported for hardware code generation. For instance, it is usually impossible to allocate memory dynamically, since memory blocks are synthesised in hardware. Another example is function recursion, which is also rarely supported for FPGA code generation.
- CPU-oriented C code may be inefficient for hardware implementations. Further details will be provided later in this chapter.
- Generated C code must be supplied with additional directives that describe FPGA implementation details.

The following set of directives is usually supported by most of the high-level synthesis tools [100, 84]:

- *Loop unrolling* implies performing iterations of a certain loop in parallel. Unrolling the loops leads to data-level parallelism.
- *Loop pipelining.* Pipelining, as a general technique, was discussed in Section 2.1.5. In relation to loops, pipelining implies overlapping iterations, i.e. starting a new iteration before finishing the previous.
- *Array partitioning* is required for algorithms in which performance is limited by memory rather than computations. Modern FPGAs mainly support single- and dual-port memory blocks, which allow reading/writing one and two pieces of data per clock cycle, accordingly. Partitioning an array into several memory blocks allows reading more than two pieces of data, which, in conjunction with replicating processing units, speeds up the algorithm.

One of the key features of SPLIT is automatic generation of the above directives.

Based on the discussion from Section 3.1.4, computations, required for implementing splitting algorithms, can be classified into vector-vector operations, matrix-vector multiplication and solving linear systems. The following sections will discuss FPGA implementation of these three classes of operations. Note that for all FPGA implementations only internal on-chip memory is used for data storage, while external memory is employed for input/output data transfer.

Vector operations

Vector operations often can be accelerated by pipelining the loop that iterates over vectors elements. For the simplest case, when consecutive iterations do not depend on each other (e.g. element-wise addition), this is implemented by supplying the source code with a pipelining directive. However, for some vector operations pipelining cannot be implemented in a straightforward manner, which is a consequence of read-write data dependencies. Consider the example of computing the sum of a vector's elements (see Listings 3.1 and 3.2).

Listing 3.1 Software-oriented C-code for calculating the sum of vector elements.

```
// SW oriented code
float vec_sum_sw(float vec_in
    [1000])
{
  int i;
  float sum 0;
  for(i = 0; i < 1000; i++)
  {
#pragma HLS PIPELINE
    sum += vec_in[i];
  }
  return sum;
}
```

Listing 3.2 Hardware-oriented C-code for calculating the sum of vector elements.

```
// HW oriented code
float vec_sum_hw(float vec_in[1000])
{
  int i,j;
  int mask[2] = {0, ~((int) 0)};
  float sum_p[8] = {0};
  for(i = 0, j = 0; i < 1000; i++) // partial
      accumulation
  {
#pragma HLS DEPENDENCE variable=sum_p array
    inter distance=8 true
#pragma HLS PIPELINE
    sum_p[j] += vec_in[i];
    j = (j+1) & mask[(j+1) != 8];
  }
  for(i = 1; i < 8; i++) // accumulation of
     the partial sums
  {
#pragma HLS UNROLL
    sum_p[0] += sum_p[i];
  }
  return sum_p[0];
}
```

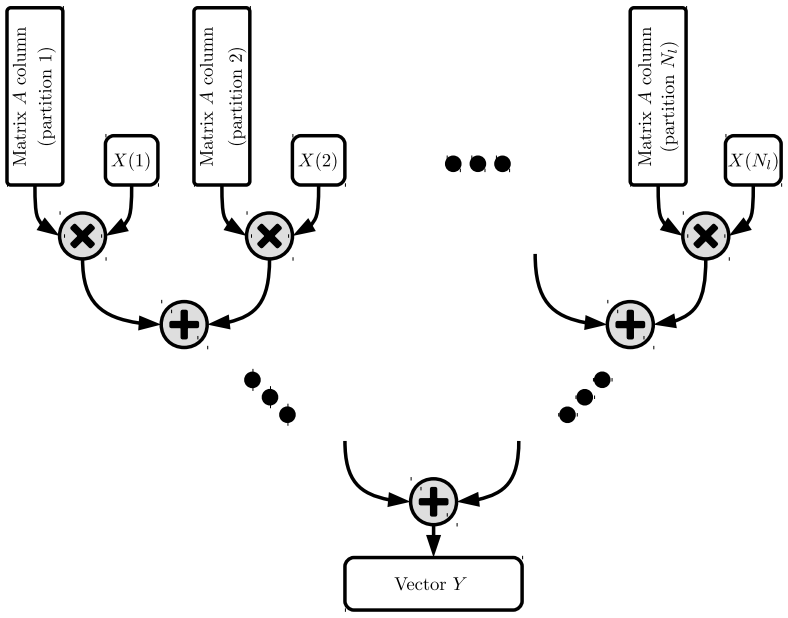


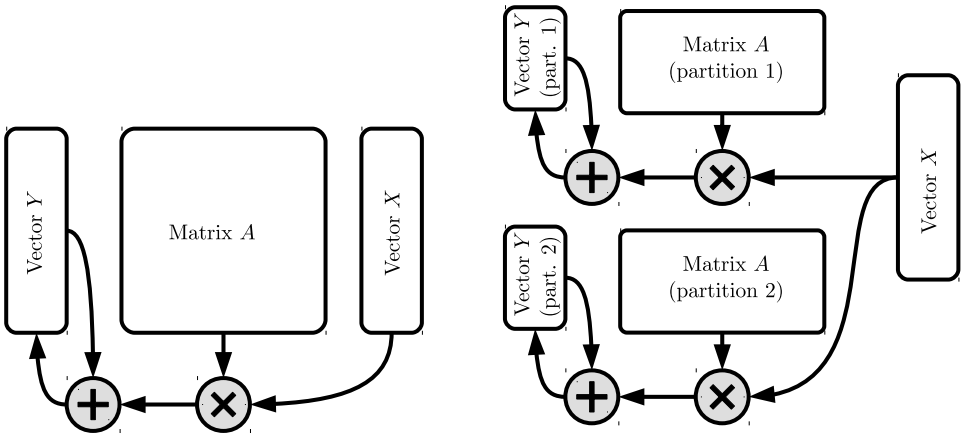
Fig. 3.2 Dense matrix-vector multiplication with an adder tree. N_l is the dimension of the square matrix.

With a software-oriented approach the next iteration of the main loop cannot be started before finishing the previous iteration due to the data reading and writing dependencies, which applies restrictions on pipelining. Hardware-oriented code computes partial sums of the vector's elements independently, which allows avoiding undesirable data dependencies and creating an efficient pipeline. After the partial accumulation is finished, the final loop accumulates the partial sums. According to the report from Vivado HLS synthesis tool, the FPGA implementation of `vec_sum_hw()` is 4.75x faster compared to `vec_sum_sw()`. Note that the same memory access pattern with the considered example holds for calculating vector norms, vector-vector and matrix-vector multiplications, which is exploited by SPLIT.

Solving linear systems

SPLIT allows solving linear systems using factorization-based approaches and offline matrix inversion. Only the latter is currently supported for FPGA code generation. Inverting a general sparse matrix leads to a dense matrix, hence the remainder of this section is focused on dense matrix-vector multiplication.

One possible way of performing dense matrix-vector multiplication is presented in the Figure 3.2. With this implementation, columns of the matrix are processed simultaneously: parallel

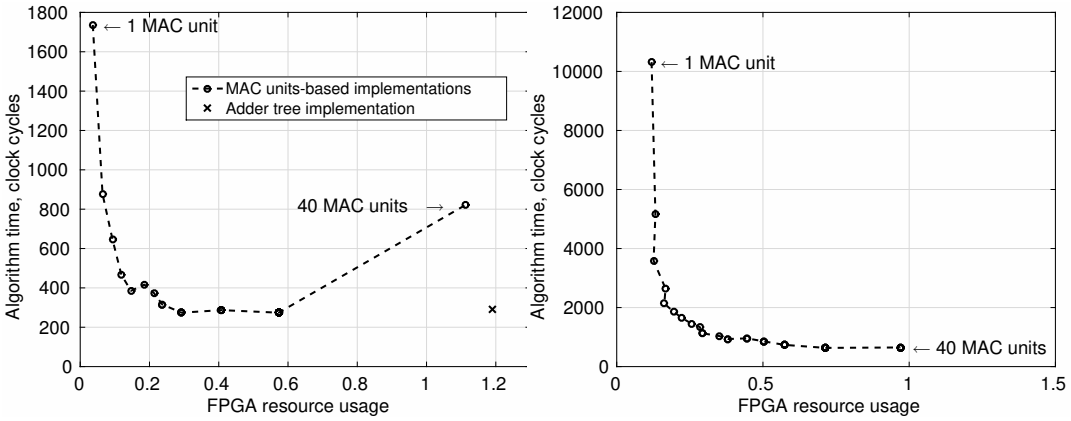


(a) One multiply-accumulate unit. (b) Two multiply-accumulate units.

Fig. 3.3 Dense matrix-vector multiplication based on multiply-accumulate units.

multiplication operation is followed by an adder tree which is connected directly to the output vector memory. Although the adder tree-based implementation has proven to be applicable for MPC applications [60, 45], it has several disadvantages. Firstly, the implementation requires complete column-wise partitioning the matrix and partitioning the input vector into separate registers, which might increase memory requirements and hence limit suitability for low cost devices. Moreover, computational resource requirements are defined by the problem dimension and cannot be varied with respect to resource availability and algorithm timing constraints. Nevertheless, the adder tree-based matrix-vector multiplication is supported by SPLIT.

SPLIT generates C code for less resource hungry realizations of matrix-vector multiplication (Figure 3.3a). This implementation does not imply partitioning the memory blocks and requires only one adder and one multiplier, which leads to extremely resource efficient but relatively slow designs. The considered combination of an adder and multiplier is known as a *Multiply-Accumulate (MAC) unit*. In order to speed up the algorithm, several MAC units can be synthesized so that different partitions of the matrix are processed in parallel (Figure 3.3b). Note that the matrix is partitioned row-wise and the number of partitions can be varied between 1 and N_{row}, where N_{row} is the number of rows. For the case when the number of MAC units is equal to the number of rows, each row will be processed separately, leading to the same memory access pattern considered in Listings 3.1 and 3.2. Hence, similarly to the example of computing the sum of vector's elements, it is required to replicate the output vector so that read/write dependencies are avoided. Once partial output vectors are calculated, the final output is computed as a sum of partial outputs. Note that



(a) Adder tree and MAC units-based implementations for a square matrix of the dimension $N_l = 40$.

(b) MAC units-based implementations for a square matrix of the dimension $N_l = 100$. Adder tree implementation did not fit into the target FPGA.

Fig. 3.4 Time vs resource usage trade-off for matrix-vector multiplication implementations with SPLIT. Points are connected to improve visibility.

summation of the partial outputs might cross out acceleration by parallelization, which will be shown by implementation examples.

Figure 3.4a demonstrates the trade off between FPGA resource usage and algorithm execution time for matrix-vector multiplication with the two approaches considered above. The target platform is the FPGA logic of a Xilinx Zynq-7000 XC7Z020 system-on-a-chip. The resource usage is measured by:

$$R_{FPGA} = \sqrt{R_{FF}^2 + R_{LUT}^2 + R_{DSP}^2 + R_{BRAM}^2}, \tag{3.13}$$

where R_{FF}, R_{LUT}, R_{DSP} and R_{BRAM} denote the relative utilization of each resource type. It can be observed from the figure that MAC units-based approach allows efficient trading off time against resource usage for the matrix 40×40. However, after a certain point introducing additional MAC units does not result in further acceleration. Moreover, a fully parallelized implementation with 40 MAC units performs considerably worse compared to most of the implementations with fewer MAC units. As discussed above, this happens because of data dependencies that result in output vector replication and additional partial output accumulation operation. According to the figure, the adder tree implementation achieves the same performance with some of the MAC unit-based implementations requiring at least twice as many FPGA resources. However, it must be taken into account that the adder three implementation returns an element of the output vector every clock cycle, which opens the possibility of building deep pipelines that involve matrix-vector

multiplication and potentially other parts of the algorithm of interest. In contrast, a MAC units-based implementation does not provide this possibility. SPLIT gives users the freedom to choose a particular approach depending on the application.

A design space exploration for a larger matrix (100×100) is presented in Figure 3.4b. Due to the limited amount of logic available on the target FPGA it was not possible to synthesize the adder tree implementation of matrix-vector multiplication. For a MAC unit-based approach the highest number of MAC units was also limited by the FPGA size; for this example the number of parallel processors was varied between 1 and 40. Note that, for example in Figure 3.4b (in contrast to the one on Figure 3.4a), the algorithm execution time is always decreasing with the number of MAC units, because the number of processors is considerably smaller than the dimension of the matrix.

Sparse matrix-vector multiplication

A detailed discussion on sparse matrix-vector multiplication in SPLIT can be found in Appendix C.

3.2 Protoip

The code generated by SPLIT is automatically deployed with Protoip. This section presents the main features of Protoip. The section is split into two parts: firstly existing features of the Protoip toolbox are described, followed by presentation of a new release of the toolbox, which is a contribution of this work.

3.2.1 Existing features

Protoip is a software tool for rapid prototyping of online optimization algorithms on FPGAs [89]. From an implementation point of view it represents a set of scripts written in Tcl, a language that is widely used for computer aided design. A Matlab interface is provided to allow working in conjunction with state-of-the-art optimization and numerical integration toolboxes. The tool supports the entire family of Xilinx Zynq-7000 devices.

The standard workflow is presented in Figure 3.5. In the first stage of the design flow the user provides C source code for the algorithm of interest, alongside synthesis directives. This code is automatically processed by Protoip using Xilinx Vivado HLS at the underlying level in order to produce a so called *intellectual property* (IP) core, a reusable integrated circuit component described

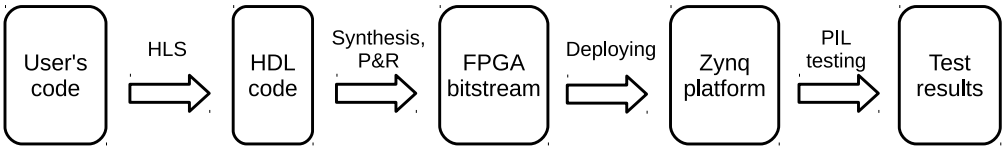


Fig. 3.5 Protoip FPGA design flow.

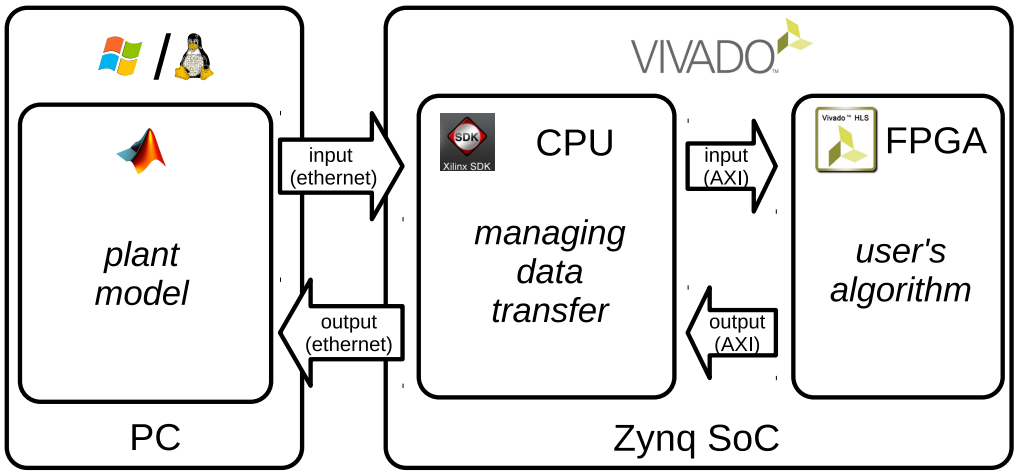


Fig. 3.6 Processor-in-the-loop test of FPGA implementations. Logos indicate software tools used by Protoip in the underlying levels.

in HDL. Following this, the IP core is synthesised using the Xilinx Vivado tool, which results in a *bitstream* that describes FPGA configuration. At the final stage, Protoip creates an environment for testing of the implemented design known as a *testbench*. Assessing the performance of an online optimization algorithm in open-loop is not a viable approach, since the main goal of an embedded platform is achieving required closed-loop performance, rather than solving a single optimization problem. For this reason it is reasonable to employ processor-in-the-loop (PIL) simulation to verify the performance of an algorithm (Figure 3.6). PIL testing allows for simulation of the controlled plant on a PC, while performing computations on a real embedded platform. In this setup the data transfer between the host PC and Zynq SoC is performed using a UDP/Ethernet interface.

3.2.2 A new release

This section presents the main features of the new release.

The previous version of Protoip produced only pure reconfigurable logic implementations, i.e. the CPU was not involved in computations. The new release adds the possibility of splitting the workload between the CPU and FPGA fabric to exploit the strengths of each subsystem and trade off

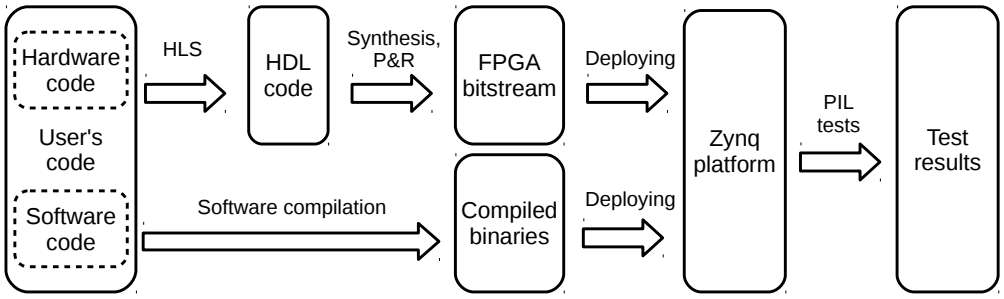


Fig. 3.7 Protoip design flow for heterogeneous implementations.

FPGA logic usage against performance. The design workflow for heterogeneous implementations consists of two parallel branches which correspond to software and hardware parts of a given algorithm, accordingly (Figure 3.7). The user provides two pieces of C code: software code for the CPU, and hardware code for the FPGA. Following this, Protoip performs circuit synthesis and software code compilation. At the final stages of design flow the implementation is deployed on a Zynq platform and a processor-in-the-loop test with a heterogeneous platform is performed.

Handling data transfer between different subsystems of a heterogeneous computing system is a crucial point that might affect overall system performance. As was briefly mentioned in Section 2.1.6, the processing system and programmable logic of a Zynq platform can exchange data via a set of AXI buses. The AXI protocol is a member of the ARM AMBA microcontroller bus family, which is an open standard for on-chip interconnects [4]. The protocol describes three types of AXI4 interface [101]:

- AXI4: High performance memory mapped interface allowing burst transactions up to 256 words per cycle.
- AXI4-Lite: A subset of AXI4, which does not support a burst-mode. Typically used for writing and reading to/from control and status registers.
- AXI4-Stream: A stream interface allowing bursts of unlimited length. Commonly used for signal processing applications.

Depending on the application, certain interfaces or combination of interfaces might be used. Protoip implements the interaction between the CPU and FPGA using AXI4 and AXI4-Lite interfaces (Figure 3.8). It should be emphasized that the CPU and FPGA do not exchange data directly; instead, they share certain areas of DDR memory. A full AXI4 interface is used for data

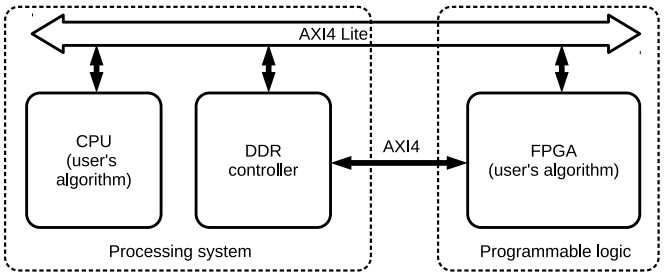


Fig. 3.8 Communication between processing system and programmable logic with Protoip.

transfer between FPGA and DDR, where data can be accessed by the CPU. Configuration and initialization is performed by CPU via an AXI4-Lite bus. Protoip provides a wrapper around these low level details so that the user can access a hardware accelerator by calling provided C/C++ functions without the need to configure control registers manually.

As shown in the Figure 3.6, a processor-in-the-loop test with the previous Protoip version requires simulation of the plant on a host PC, which requires the sending/receiving of data over the UDP/Ethernet interface at every sampling instant. Simulating the plant on the internal CPU within a SoC, which becomes possible in the new release, reduces data transfer time considerably by using on chip AXI interface instead of external UDP. According to simulation results for an algorithm with one input vector and one output vector (100 elements each) the communication time is reduced by a factor of 1500.

Desktop PC to Zynq communication speed and reliability might also be critical for processor-in-the-loop simulations. The previous release of Protoip relied on a UDP/Ethernet interface, which is a common and relatively fast protocol, however with no transmission guarantees. Lack of reliability might be crucial for design space space exploration, where processor-in-the-loop simulations may run autonomously for several weeks. A possible solution to this problem is to employ a PCI interface, which has proven to be a reliable solution for FPGA to PC communication [49]. However, the PCI interfaces are rarely available on low-cost prototyping boards. For this reason it was decided to keep using UDP/Ethernet so that no additional hardware is introduced. To improve reliability, another communication layer with a retransmission mechanism was implemented on the top of UDP. As a result, communication reliability was improved without changing hardware requirements.

Another feature of the release is a new way of managing Vivado, Vivado HLS and Xilinx SDK projects. In particular:

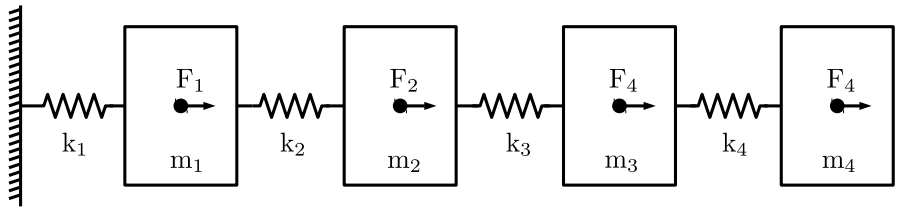


Fig. 3.9 Mass-spring system.

- Users get flexibility of managing projects (e.g. adding new libraries, project parameters). With the previous version of Protoip only a predefined set of project parameters was supported.
- A possibility of handling multiple-file C/C++ projects is added. This makes Protoip more friendly in relation to code generation tools.

In conclusion, the new Protoip release features, i.e. contributions of this work, are:

- The possibility of prototyping heterogeneous implementations.
- A retransmission mechanism on the top of UDP/Ethernet for reliable processor-in-the-loop simulations.
- Improving the way of managing application projects to allow working in conjunction with code generation tools.

3.3 Numerical results

In this work we use a Xilinx Zynq-7000 XC7Z020 SoC with dual-core ARM Cortex-A9 and FPGA logic, which contains: 53200 LUTs, 106400 FFs, 220 DSP blocks and 140 block RAMs with total capacity 4.9 Mb. A high throughput-oriented AXI bus provides fast communication between the CPU and FPGA and hence allows heterogeneous implementations of computationally intensive algorithms. For all considered scenarios the CPU clock frequency was set to 667 MHz, while the FPGA was clocked at 150 MHz.

We consider the optimal control problem of the form (3.7) in relation to the chain of masses illustrated in Figure 3.9. The system of interest has 8 states (positions and velocities of the masses) and 4 inputs (forces applied to the masses). The masses are equal to 1 kg and the spring constants have a value of 1 Nm^{-1}. The OCP has the following parameters: $Q_d = I$, $R_d = I$, $P_d = Q_d$, sampling period is $T_s = 0.5$, zero-order hold is assumed. The following constraints are considered:

Table 3.1 Benchmark problems

	Horizon length, N	Linear system dimension	Number of MAC units for solving linear systems on the FPGA
Design 1	5	140×140	16
Design 2	6	164×164	18
Design 3	7	188×188	21
Design 4	8	212×212	24
Design 5	9	236×236	26
Design 6	10	260×260	29

- Box constraints on the inputs with lower and upper bounds of -0.5 and 0.5 accordingly.
- Soft box constraints on the states (masses positions) with lower and upper bounds of -0.5 and 0.5 accordingly. Softening is performed as described in [45].

Other design parameters are presented in Table 3.1. Note that the number of MAC units for solving linear systems (i.e. dense matrix-vector multiplication) on the FPGA is proportional to the linear system dimension, which keeps the relative degree of parallelism constant. We compile several MPC implementations using the AMA algorithm on the underlying level:

- Pure FPGA implementation: all numerical operations are performed on the FPGA within this implementation. As was noted previously, the linear system solve operation is based on offline inversion and online dense matrix-vector multiplication. We denote the implementation as **HW**.
- Pure FPGA implementation with CPU-oriented C code without synthesis directives: in contrast to the above implementation, the C code is not supplied with synthesis optimization directives. We denote the implementation as **HW-unoptimized**.
- Heterogeneous implementation: Within this implementation all the operations, except solving linear systems, are computed on a single core of ARM Cortex-A9 processor and the linear system is solved on the FPGA. This implementation is denoted as **HG**.
- Pure software implementation with sparse LDL factorization-based linear solver: the implementation implies performing all numerical operations on a single core of ARM Cortex-A9 processor. Denoted as **SW-LDL**.

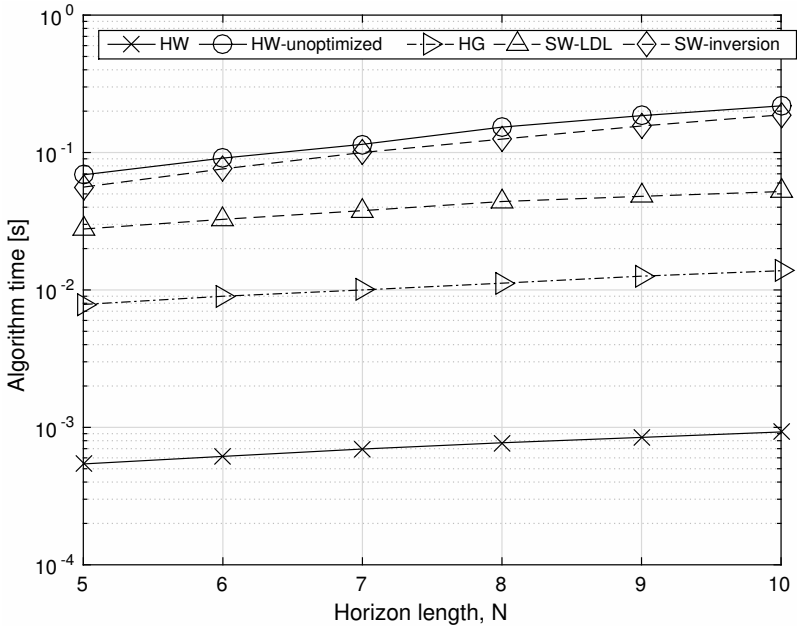
- Pure software implementation with offline inversion-based linear solver: the implementation implies performing all numerical operations on a single core of ARM Cortex-A9 processor. Denoted as **SW-inversion**.

All the implementations above use single precision floating-point arithmetic. For FPGA-based implementations (**HW**, **HW-unoptimized** and **SW-inversion**) dense matrix-vector multiplication is performed using MAC units-based approach (Section 3.1.5).

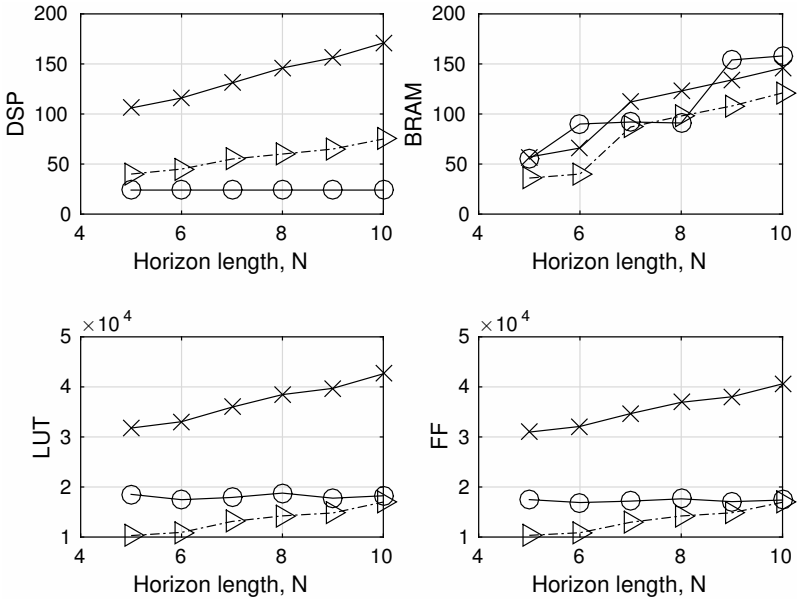
As illustrated in Figure 3.10, the FPGA implementation with synthesis directives (**HW**) significantly outperforms all other designs, with the maximum speedup over the **SW-LDL** implementation of $\times 56$. The second fastest implementation, **HG**, outperforms **SW-LDL** by a factor of 4. This is explained by the fact that **HW** and **HG** implementations exploit the parallelization potential of the linear system solve operation. In contrast, a sparse LDL factorization-based approach, which is based on offline factorization and online forward-backward solve, cannot benefit from parallelism due to the lack of definite pattern in-fill and hence is implemented on the CPU. However, for sequential software implementations, sparse factorization outperforms dense matrix-vector multiplication, which can be confirmed by comparing **SW-LDL** and **SW-inversion**. Finally, it can be observed from the figure that compiling C code for FPGA without special synthesis detectives (**HW-unoptimized** implementation) leads to poor results, which justifies using a dedicated hardware code generation tool.

As discussed in Section 3.1.5, SPLIT allows controlling algorithm time and resource usage scaling by varying the number of MAC units for dense matrix-vector multiplication. Consider the trade-off for **HW** implementation of Design 1 from Table 3.1, where the number of MAC units is varied between 1 and 30 (Figure 3.11). It can be observed that increasing the number of processors improves algorithm time at the cost of using more resources. When selecting the number of parallel processors for a particular application, one has to keep in mind Ahmdal's law, which states that overall algorithm parallelization speedup is limited by the sequential part of the algorithm. This explains why, after reaching a certain point, parallelizing computations does not lead to significant speedups.

Comparing FPGA implementations of optimization algorithms presented in the literature is not straightforward, due to the presence of many algorithmic and hardware level design parameters. Table 3.2 attempts to compare the Design 1 implementation against tailored ADMM-based FPGA



(a) Algorithm time scaling with the horizon length.



(b) Resource scaling with the horizon length.

Fig. 3.10 Time and resource scaling of software, heterogeneous and hardware implementations. Timing results are presented for 40 iterations of the AMA algorithm.

implementation of predictive control [45], which also uses a mass-spring system model. Some remarks on the presented comparison:

- The implementation in [45] is tailored for optimal control problems of the form (3.7), while the this thesis implementation handles a general convex optimization problem of the form (3.1), considering (3.7) as one possible application.

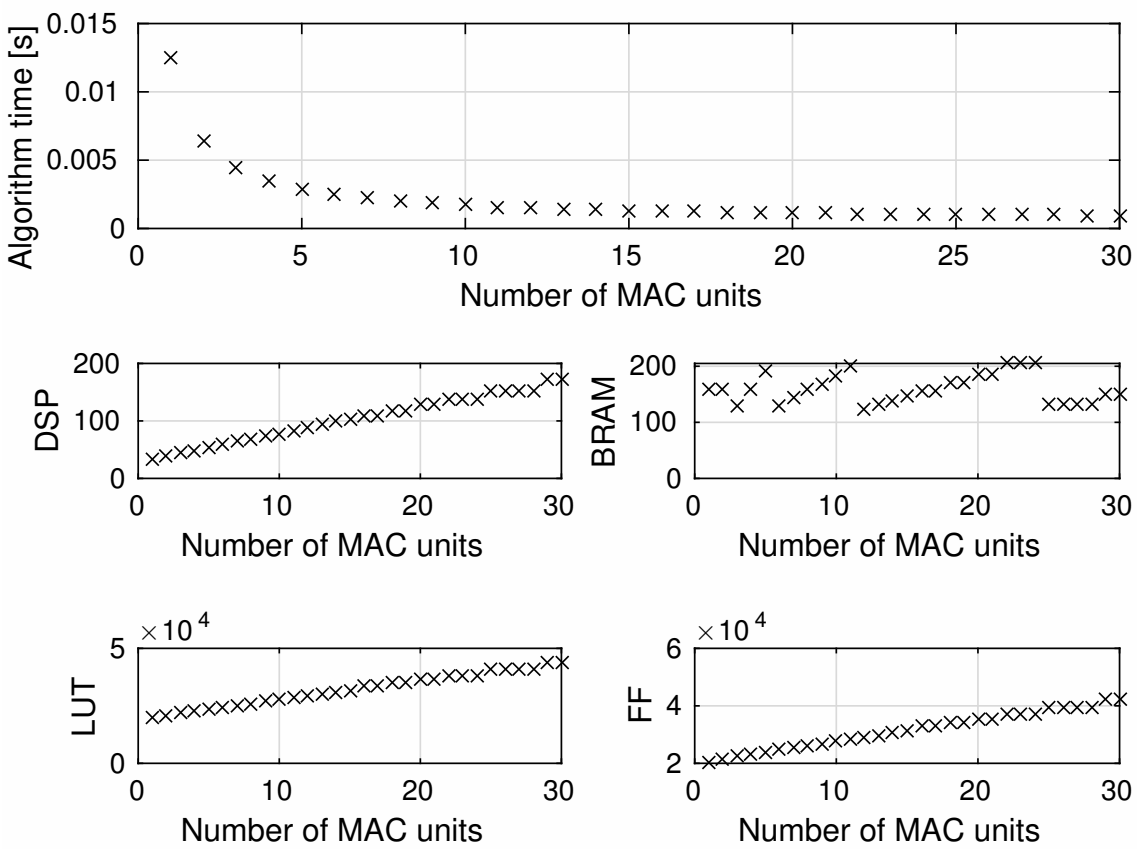


Fig. 3.11 Trading off resources usage against latency by varying the number of MAC units.

- This thesis implementation targets lows cost FPGA from the Xilinx Artix-7 family, while [45] is implemented on a high peformance platform from Xilinx Virtex-6 family.
- Provided that linear systems are solved using offline matrix inversion, the AMA algorithm that is used in this work has an identical computational pattern with ADMM, which is used in [45] (see Section 3.1.4), hence the comparison between implementations is fair.

3.4 Summary and future work

This chapter presented a code generation tool for hardware and heterogeneous implementations of embedded optimization using operator splitting methods. Special attention has been paid to giving users the possibility of changing the degree of parallelism to enable tailoring implementations with respect to the design objectives and available computational resources. Experimental results have confirmed that automatically generated hardware implementations of optimization-based controllers significantly outperform corresponding CPU realizations, which opens new possibilities

Table 3.2 FPGA implementations of MPC for mass-spring systems.

Implementation	Platform	Clock freq., MHz	Number representation	Programming language	Model	Algorithm	Horizon, N	Constraints	Number of iterations	Algorithm time, μs	Fixed point multipliers	DSP units
[45]	Xilinx Virtex-6 LX-75	400	Fixed-point	VHDL	8 states 4 inputs	ADMM	10	4 input; 4 input rates; 4 soft state	40	23.4	216	N/A
HW, this thesis	Xilinx Artix-7 XC7Z020	150	Single floating-point	C Vivado HLS	8 states 4 inputs	AMA	10	4 input; 4 soft state	40	926	N/A	171

of using FPGA-based control for engineers with various backgrounds. However, the comparison with a tailored MPC implementation on an FPGA [45] shows that there is a large performance gap between automatically generated and tailored fixed-point arithmetic implementations.

Further work will be focused on:

- Implementing splitting algorithms with fixed point arithmetic to achieve better performance vs resource usage trade-offs.
- Incorporating systematic techniques for automating the selection of design parameters, e.g. the degree of parallelism. This will allow avoiding situations illustrated as in Figure 3.11, where starting from a certain point, parallelizing computations leads to negligible improvement due to Ahmdal's law.

Chapter 4

Nonlinear Predictive Control on a Heterogeneous Computing Platform

This chapter presents a heterogeneous hardware implementation of a nonlinear MPC controller based on an interior-point algorithm. The proposed implementation provides flexibility of splitting the workload between a general-purpose CPU with a fixed architecture and a field-programmable gate array. This allows for trading off contradicting design objectives, namely control performance and computational resource usage. A new way of exploiting the structure of the Karush-Kuhn-Tucker (KKT) matrix yields significant memory savings, which is crucial for reconfigurable hardware. For the considered case study, an 18x memory saving compared to existing approaches and a 36x speedup over a software implementation with ARM Cortex-A9 processor are reported.

4.1 Hardware acceleration of NMPC: main challenges

Recent developments in reconfigurable computing allowed acceleration of predictive control algorithms on field-programmable gate arrays, which resulted in low-cost and resource-efficient realizations of custom quadratic programming solvers for MPC, e.g. [45, 38] and Chapter 3 of this thesis. Extending a hardware acceleration approach from linear to nonlinear model predictive control (NMPC), which can be considered as the next logic step, requires mapping numerical integration algorithms on hardware, which is not a trivial task, since dynamic models that describe the physical world are problem-dependent. As a result, instead of using systematic dynamic optimization, existing FPGA implementations of NMPC either rely on stochastic optimization [103]

or approximate the offline solution with machine learning techniques [10]. These approaches cannot guarantee scalability nor closed-loop performance.

Accelerating deterministic algorithms on hardware might be achieved by employing heterogeneous computing platforms that involve both a general-purpose processor with a fixed architecture and FPGA logic. For example, [74] presents a heterogeneous implementation of a multiple-shooting based NMPC algorithm. The authors propose implementing integration in software while accelerating a fast gradient-based quadratic programming solver on an FPGA. The reported speedup of the heterogeneous implementation over a software realization is 1.6x and further improvement is limited, since integration and optimization algorithms have comparable computational complexity. This is a consequence of Amdahl's law [2], which states that an algorithm's speedup is limited by the part of the workload that cannot benefit from acceleration.

We present a new heterogeneous implementation of nonlinear interior-point algorithm for predictive control. The main features of the proposed implementation are:

- A method for scheduling sparse matrix-vector multiplication within an iterative linear system solvers to enable significant improvements in terms of computation time vs resource usage. For the example considered, an 18x memory saving compared to existing approaches and a 36x speedup over a software implementation are reported.
- Flexible splitting of the algorithmic workload between software and hardware for trading off the computational resource usage against performance.

4.2 Optimal control problem formulation and temporal discretization

We consider nonlinear time-invariant systems that can be described as an ordinary differential equation (ODE) of the form

$$\dot{x}(t) = f_c(x(t), u(t)), \tag{4.1}$$

where $f_c : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^n$. We consider the nonlinear optimal control problem (OCP) with initial state $\hat{x}$ and prediction horizon T:

$$\min_{x,u,s} \quad \int_0^T ||h(x(t), u(t), s(t))||_2^2 dt + ||h_T(x(T), s_T)||_2^2 \tag{4.2a}$$

$$
\begin{aligned}
\text{subject to:} \quad & x(0) = \hat{x}, && && \text{(4.2b)}\\
& \dot{x}(t) = f_c(x(t), u(t)), && \forall t \in [0, T] && \text{(4.2c)}\\
& q(x(t), u(t), s(t)) = 0, && \forall t \in [0, T] && \text{(4.2d)}\\
& q_T(x(T), s_T) = 0, && && \text{(4.2e)}\\
& x_l \leq x(t) \leq x_u, && \forall t \in [0, T] && \text{(4.2f)}\\
& u_l \leq u(t) \leq u_u, && \forall t \in [0, T] && \text{(4.2g)}\\
& s_l \leq s(t) \leq s_u, && \forall t \in [0, T] && \text{(4.2h)}\\
& x_{Tl} \leq x(T) \leq x_{Tu}, && && \text{(4.2i)}\\
& s_{Tl} \leq s_T \leq s_{Tu}, && && \text{(4.2j)}
\end{aligned}
$$

where $h : \mathbb{R}^n \times \mathbb{R}^m \times \mathbb{R}^{n_S} \to \mathbb{R}^{n_h}$ and $h_T : \mathbb{R}^n \times \mathbb{R}^{n_{ST}} \to \mathbb{R}^{n_{hT}}$. Slack trajectory s and slack variable s_T are introduced alongside with equations (4.2d) and (4.2e), which is a common technique that allows for the handling of general nonlinear inequality constraints [97]. Note that in the general case $s(T) \neq s_T$. The presented formulation can be generalized for time-varying reference tracking, which, as will be shown later, requires only changing the software part of the algorithm.

Direct solution of the continuous-time optimal control problem (4.2) involves two main stages: integration, i.e. solving the ordinary differential equation (ODE), and optimization. Implementing integration on an FPGA is not desirable because of the following reasons:

- The ODE (4.2c) may involve mathematical expressions that in contrast to standard addition and multiplication operations, require significant amounts of computational resources (Figure 2.6) and might be unsuitable for pipelining.
- A vector function f_c is composed of scalar functions that might have different underlying mathematical operations and different evaluation complexities, i.e. f_c might have *irregular* structure. Irregularity potentially limits reusing computational logic and speedup by parallelization.

Optimization algorithms, on the other hand, can benefit from hardware acceleration due to (i) their iterative nature, which is beneficial for reusing computational logic, and (ii) the fact that underlying linear algebra algorithms can be efficiently mapped onto hardware [44].

Taking the above into account we consider two classes of algorithms for solving (4.2): shooting-based and direct transcription algorithms [15]. The common feature of shooting methods is decoupling the ODE and optimization solvers. Accelerating only the latter does not result in significant improvements, due to Amdahl's law, since the workloads of the two operations are comparable. In contrast, direct transcription algorithms transform (4.2) directly to a discrete OCP by approximating the ODE with algebraic equations based on numerical integration equations, i.e.

$$\min_{\substack{x_0,\ldots,x_N \\ u_0,\ldots,u_{N-1} \\ r_0,\ldots,r_{N-1} \\ s_0,\ldots,s_{N-1},s_T}} \sum_{k=0}^{N-1} \left(T_s ||h(u_k, x_k, s_k)||_2^2 \right) + ||h_T(x_N, s_T)||_2^2 \tag{4.3a}$$

subject to: (4.3b)

$$x_0 = \hat{x}, \tag{4.3c}$$

$$c(x_k, u_k, r_k) = \begin{bmatrix} x_{k+1} \\ 0 \end{bmatrix}, \qquad k = 0, \ldots, N-1 \tag{4.3d}$$

$$q(x_k, u_k, s_k) = 0, \qquad k = 0, \ldots, N-1 \tag{4.3e}$$

$$q_T(x_N, s_T) = 0, \tag{4.3f}$$

$$x_l \leq x_k \leq x_u, \qquad k = 0, \ldots, N-1 \tag{4.3g}$$

$$u_l \leq u_k \leq u_u, \qquad k = 0, \ldots, N-1 \tag{4.3h}$$

$$s_l \leq s_k \leq s_u, \qquad k = 0, \ldots, N-1 \tag{4.3i}$$

$$x_{Tl} \leq x_N \leq x_{Tu}, \tag{4.3j}$$

$$s_{Tl} \leq s_T \leq s_{Tu}, \tag{4.3k}$$

(4.3l)

where N and T_s, respectively, denote the horizon length and the sampling time, $r_k = \begin{bmatrix} r_k^{(1)T} & \cdots & r_k^{(l)T} \end{bmatrix}^T \in \mathbb{R}^{nl}$ is a vector of integrator intermediate stages and l is the number of integrator stages per sampling instant.

With a trapezoidal integrator, (4.3d) would be given by:

$$x_k + T_s\left(r_k^{(1)} + r_k^{(2)}\right)/2 = x_{k+1} \tag{4.4a}$$

$$r_k^{(1)} - f_c(x_k, u_k) = 0 \tag{4.4b}$$

$$r_k^{(2)} - f_c\left(x_k + T_s\left(r_k^{(1)} + r_k^{(1)}\right)/2, u_k\right) = 0 \tag{4.4c}$$

The OCP (4.3) can be transformed into an NLP of the form

$$\min_{\theta} \quad \frac{1}{2}||f(\theta)||_2^2 \tag{4.5a}$$

$$\text{subject to} \quad p(\theta) = 0, \tag{4.5b}$$

$$J\theta - d \leq 0, \tag{4.5c}$$

where $\theta = [x_0^T\ u_0^T\ r_0^T\ s_0^T \ldots x_{N-1}^T\ u_{N-1}^T\ r_{N-1}^T\ s_{N-1}^T\ x_N^T\ s_T^T]^T$. Note that J is a matrix of zeros and positive/negative ones since (4.3) contains only bound inequalities.

NLP (4.5) incorporates both integration and optimization, which potentially opens the possibility of accelerating both subproblems. Primal-dual interior-point algorithms have proven their efficiency for the numerical solution of optimal control problems [83]. Moreover, with interior-point algorithms, the structure of the KKT matrix associated with the OCP remains fixed (unlike with active set methods), which is desirable for hardware implementations [44].

The next section introduces a structure-exploiting heterogeneous implementation of an interior-point algorithm for solving (4.5). The implementation supports the use of Runge-Kutta family integrators for temporal discretization, considering Butcher tableau as a design parameter. However, results for memory saving, algorithm acceleration and closed-loop simulations are obtained with a trapezoidal integrator (4.4), which belongs to the family of explicit integrators and hence allows efficient handling of stiff ODEs.

4.3 Algorithm and implementation details

4.3.1 Primal-dual interior-point algorithm

Interior-point algorithms solve NLPs by incorporating inequality constraints into the objective function using a logarithmic barrier function scaled with a barrier parameter. The modified problem

Algorithm 4 Primal-dual interior-point method for NLP

Initial point $[\theta_{[0]}^T, \nu_{[0]}^T, \lambda_{[0]}^T]^T : \lambda_{[0]} > 0, G\theta_{[0]} - d < 0$
Reduction parameter $0 < \sigma \leq 1$
for $k = 0$ to n_{iter} **do**
    $A_{[k]} = \begin{bmatrix} H_{[k]} + GW_{[k]}G^T & \nabla_\theta p^T(\theta_{[k]}) \\ \nabla_\theta p(\theta_{[k]}) & 0 \end{bmatrix}$
    $b_{[k]} = \begin{bmatrix} r_{dual} \\ r_{eq} \end{bmatrix}$
    Solve $A_{[k]} z_{[k]} = b_{[k]}$ for $z_{[k]} = \begin{bmatrix} \Delta\theta_{[k]} \\ \Delta\nu_{[k]} \end{bmatrix}$
    $\Delta\lambda_{[k]} = -\Lambda_{[k]} e - G^{-1}(\theta_{[k]})\mu e - G^{-1}(\theta_{[k]})\Lambda_{[k]}\nabla_\theta g(\theta_{[k]})\Delta\theta_{[k]}$
    $\alpha_{[k]} = \max_{(0,1]} \alpha : \lambda_{[k]} + \alpha_{[k]}\Delta\lambda_{[k]} > 0, G(\theta_{[k]} + \Delta\theta_{[k]}) - d < 0$
    $[\theta_{[k+1]}^T, \nu_{[k+1]}^T, \lambda_{[k+1]}^T]^T = \alpha_{[k]}[\Delta\theta_{[k]}^T, \Delta\nu_{[k]}^T, \Delta\lambda_{[k]}^T]^T + [\theta_{[k]}^T, \nu_{[k]}^T, \lambda_{[k]}^T]^T$
end for

is solved by performing consecutive Newton steps with or without globalization. Algorithm 4 and Appendix B.1 outline a primal-dual interior-point algorithm for solving (4.5). The algorithm computes a barrier parameter based on the current complementary value and reduction parameter, which is one of the simplest and most common approaches [66]. Furthermore, there is no globalization strategy incorporated in the algorithm. This choice is justified by the fact that globalization algorithms involve repetitive evaluation of $f(\theta)$ and $p(\theta)$, which, as was previously mentioned, often have irregular structure and hence discourage efficient acceleration. An extensive comparative study of barrier update strategies and globalization techniques for nonlinear interior-point methods can be found in [66].

Another algorithmic choice that requires justification is the Hessian approximation method. Most of existing interior-point algorithms for large scale optimization use either Broyden-Fletcher-Goldfarb-Shanno (BFGS) [27] or Gauss-Newton approximators. The BFGS algorithm, if applied blockwise, can achieve a block diagonal structure for the Hessian approximation matrix when used for optimal control applications [41]. The Gauss-Newton algorithm, in addition, preserves sparsity within each block, which is particularly beneficial taking into account scratchpad memory limitations of reconfigurable computers. However, memory savings come at the price of narrowing the application scope to least-squares types of problems. In this work we accept this limitation and use the Gauss-Newton approach.

The main workload of the algorithm is concentrated in solving the system of linear equations. The matrix associated with the problem is symmetric and can be reordered to achieve the structure presented in Figure 4.1. A symmetric system of linear equations can be solved using direct methods,

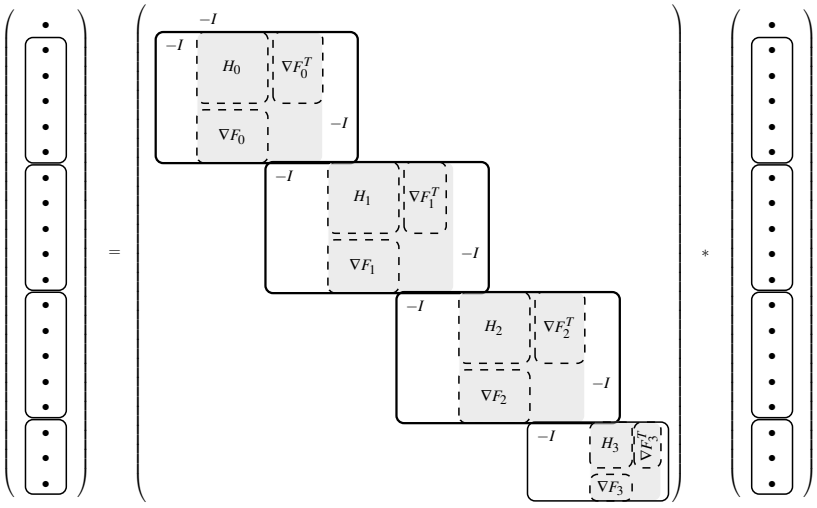


Fig. 4.1 Structure of the KKT matrix assosiated with the problem (4.5). H_i and ∇F_i denote Hessians and Jacobians of the corresponding sampling instant in the prediction horizon.

e.g. LDL decomposition, or iterative methods, e.g. Minimum Residual (MINRES) algorithm [71]. Decomposition algorithms often involve many division computations, which are more complicated from a hardware implementation point of view compared to addition and multiplication. Moreover, parallelizing computations is not straightforward with direct algorithms. In contrast, iterative methods mainly rely on matrix-vector multiplications, while having very few division and square root computations. In this work we use the MINRES algorithm, which can be efficiently mapped onto hardware [16] and is well studied in relation to optimal control applications [83]. The algorithm performs minimization of the residual $||b - Az||_2$ over a Krylov subspace, which is constructed iteratively by the Lanczos kernel [71]. The Lanczos kernel is based on a three-term recurrence, hence there is no need to store the entire Krylov subspace (unlike with the generalized minimal residual method [78]). From an implementation point of view, the MINRES algorithm is mostly based on addition and multiplication operations, requiring only two scalar divisions and two scalar square root computations per iteration.

4.3.2 Proposed implementation

Algorithm 4 with the MINRES linear solver can be visualized with the block diagram in Figure 4.2. We consider four different ways of splitting the workload between software and hardware:

- The entire algorithm is implemented in software, i.e. ARM Cortex-A9 processor. We denote this implementation by SW.

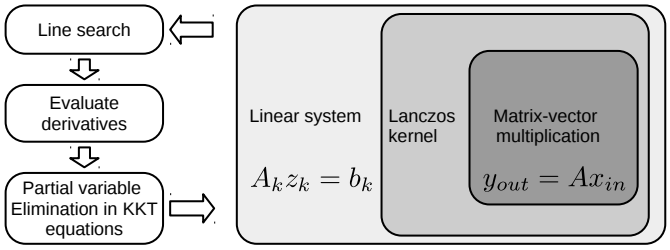


Fig. 4.2 Algorithm 4 flow diagram with MINRES solver.

- Only the matrix-vector multiplication is accelerated in hardware and the rest is implemented in software (HG_1).
- The whole Lanczos kernel is accelerated in hardware and the rest is implemented in software (HG_2).
- The whole linear system solver is accelerated in hardware and the rest is implemented in software (HG_3).

Note that only linear system solver-related operations are accelerated in hardware. This is explained by the iterative nature of the solver, which is desirable for computational logic reusage. For all considered options, the Jacobians evaluations are implemented in software to avoid synthesising resource-consuming nonlinear operators. Operations that are implemented in hardware can be classified into:

- *Scalar operations*, which do not require acceleration.
- *Vector-vector operations*, which can be efficiently pipelined in hardware.
- *Matrix-vector multiplication*, which is the most resource-consuming part. Efficient implementation requires exploiting sparsity.

One possible way to exploit the structure of A_k is based on considering the matrix as banded [44]. In this case the number of parallel computations is defined by the bandwidth and cannot be changed with respect to resource availability. In this work we treat the matrix as block sparse and, in addition, exploit sparsity within each block, which can result in an 18x saving in memory (Figure 4.3).

Matrix-vector multiplication can potentially be parallelized by simultaneous processing of the solid line blocks in Figure 4.1, which correspond to different sampling instants. However, consecutive blocks are coupled, i.e. each block accesses areas in the input vector associated with its

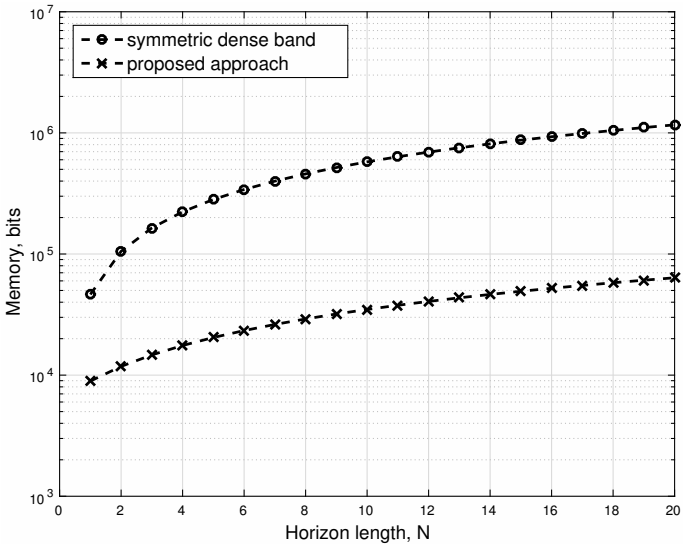


Fig. 4.3 Memory requirements for the storing matrix on Figure 4.1. ODE model and OCP parameters are presented in Section 4.4, trapezoidal method is used for ODE integration. Details on dense band implementation can be found in [17]. Actual memory savings may vary depending on the memory block size of a particular computing platform.

neighbours, which results in restrictions on memory partitioning. It can be noted that blocks are coupled only via negative identity matrices. For a matrix-vector multiplication, handling negative identity matrices is reduced to data transfer operations (with changing the sign bit) and does not require any arithmetic operators. After negative identity matrices are processed, the remaining parts (grey area) can be parallelized, since there is no overlap in accessing input vector partitions.

The efficiency of multiplication also depends on how sparsity within each block is handled. For example, consider the sparse matrix in Figure 4.4. Each non-zero element in the matrix corresponds to a multiply-accumulate (MAC) operation, i.e. each non-zero element is multiplied by a corresponding element of the input vector and added to the corresponding element of the output vector. Although the order of processing non-zero elements with floating point arithmetic might affect the final result, in practice this effect is assumed to be negligible. More importantly, this order has an impact on output vector read-write dependencies and hence pipelining possibilities. For the considered example, processing of a_{12} cannot be started before a_{11} is fully processed, since both elements require writing data to the same element of the output vector. Hence, MAC operations have to be scheduled in such a way that the distance (in terms of computer clock cycles) between processing non-zero elements from the same row is maximized. This scheduling problem can be

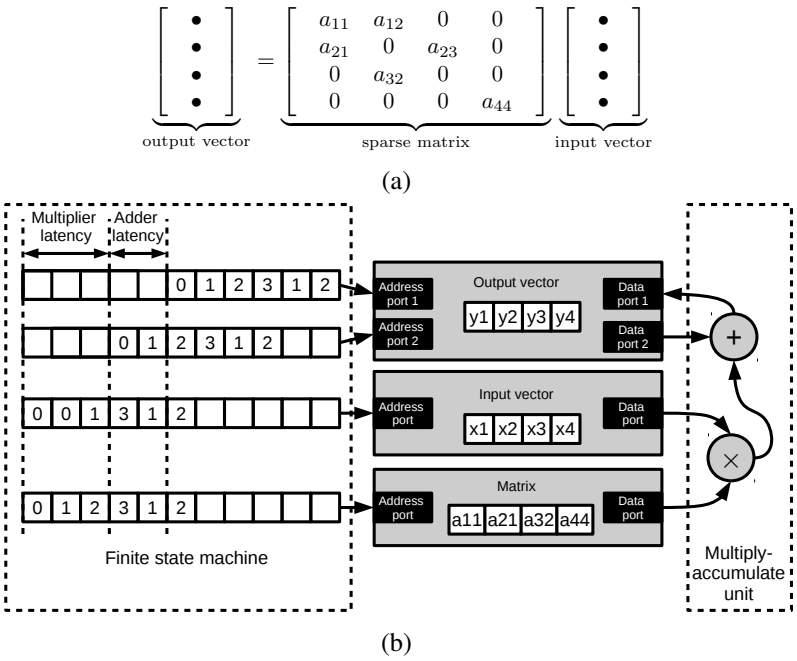


Fig. 4.4 An example of a sparse matrix (a) and the corresponding circuit (b) for matrix vector multiplication.

formulated as an optimization problem

$$\max_{d,S_{sched}} \quad d \tag{4.6a}$$

subject to:

$$|t(a_{ij}) - t(a_{kl})| > d \qquad \forall a_{ij} \in M, \forall a_{kl} \in M : i = k, j \neq l \tag{4.6b}$$

$$1 \leq t(a_{ij}) \leq N_{nz} \qquad \forall a_{ij} \in M \tag{4.6c}$$

$$t(a_{ij}) \neq t(a_{kl}) \qquad \iff i \neq k, j \neq l \tag{4.6d}$$

where $d \in \mathbb{Z}$ is a slack variable, $M = \{a_{11}, a_{12}, a_{21}, a_{23}, a_{32}, a_{44}\}$ is the set of non-zero elements, N_{nz} is the number of non-zero elements, $t(a_{ij})$ is the number of time instant, at which processing of a_{ij} starts, and $S_{sched} = [t(a_{11}), t(a_{12}), t(a_{21}), t(a_{23}), t(a_{32}), t(a_{44})]^T \in \mathbb{Z}^{N_{nz}}$. Since all blocks in Figure 4.1 have the same sparsity pattern, the scheduling problem is solved only once during design. The resulting MAC circuit for the considered example is shown in Figure 4.4. Note that due to symmetry of A_k only the lower triangular part is stored.

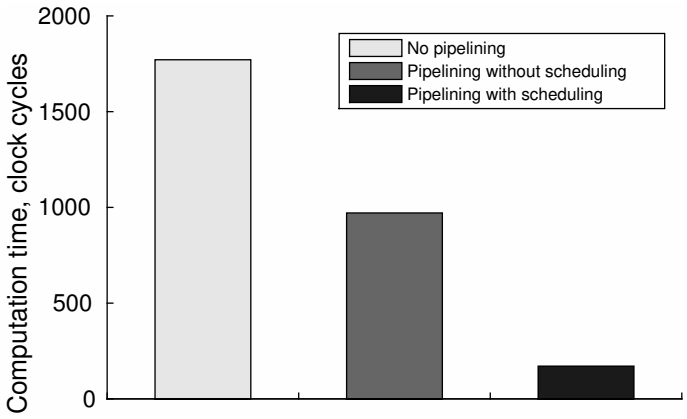


Fig. 4.5 The impact of scheduling on computation time of a sparse matrix-vector multiplication (for each gray block in Figure 4.1). It is assumed that the latency for the addition and multiplication units are 6 and 5 clock cycles, respectively. The number of nonzero elements is 161, which is the case for the example described in Section 4.4.

The impact of scheduling on the time taken for sparse matrix-vector multiplication is presented in Figure 4.5. For this case study, problem (4.6) was solved using the YALMIP built-in branch and bound solver [56] [1]. If pipelining is implemented without scheduling, the *initiation interval* is equal to the adder latency, which limits potential improvement. Initiation interval is defined as the number of clock cycles that elapses between starting processing two successive non-zeros. With a systematic scheduling approach, read-write dependencies are handled optimally, which allows one to start processing a new non-zero element after the previous as quickly as possible, i.e. achieving the minimal initiation interval. For the ODE model presented in Section 4.4 it was possible to achieve initiation interval of one clock cycle.

Although matrix-vector multiplication can be parallelized by allocating a MAC unit to each grey block (Figure 4.1), we propose trading off computation time against resource usage by varying the number of MAC units, as shown in Figure 4.6. The number of MAC units is denoted by P.

4.3.3 Limitations of the proposed implementation

We highlight certain limitations of the proposed implementation:

- For matrix-vector multiplication, input and output vectors have to be partitioned in accordance with the matrix Figure 4.1, which, depending on the parallelism level P, might increase FPGA block memory usage. This issue can be partially addressed by placing corresponding

[1] A custom algorithm for solving (4.6) is presented in Appendix C.

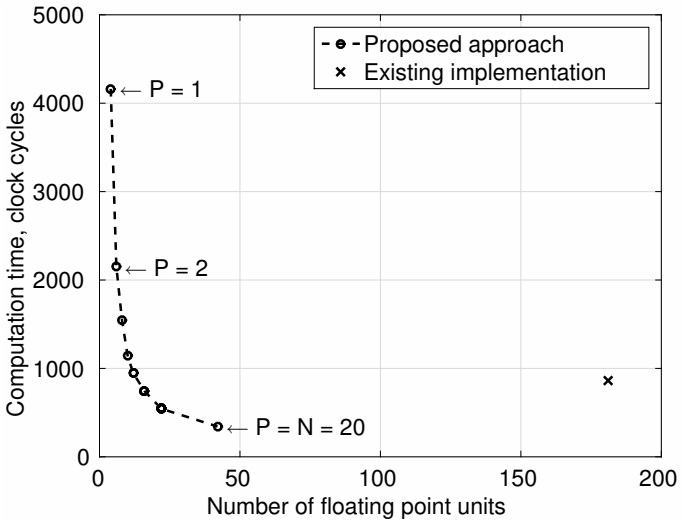


Fig. 4.6 Trading off computational time against processor resource usage for matrix-vector multiplication (Figure 4.1). Each circle represents a design with a certain degree of parallelism. The floating point unit is either floating point addition or multiplication. For the considered test case, $N = 20$ and the ODE model is given in Section 4.4. It is assumed that the latency of the addition and multiplication units are 6 and 5 clock cycles, respectively. Details on the existing implementation can be found in Boland and Constantinides [17].

partitions of the matrix and input vector in the same memory block, which will not affect performance, provided the computing platforms supports dual-port memory blocks. Another possible solution is implementing vector partitions using lookup tables (provided partitions are sufficiently small) instead of using block memory.

- In contrast to [17], our implementation requires control logic, which might limit FPGA clock frequency. However, matrix-vector multiplication, being a part of a larger algorithm, is often not the main bottleneck for increasing clock frequency [16].

- Applying a preconditioner directly to original linear system may destroy in-block sparsity and hence eliminate memory savings. To overcome this limitation, preconditioners can be applied indirectly by performing two matrix-vector multiplications per MINRES iteration so that sparsity patterns are preserved, both for preconditioner and the original matrix [36]. In this work we use a sparsity-preserving prescaler described in [42]. Although preserving sparsity, this prescaler changes negative identity matrices into general diagonal matrices, which slightly complicates the algorithm.

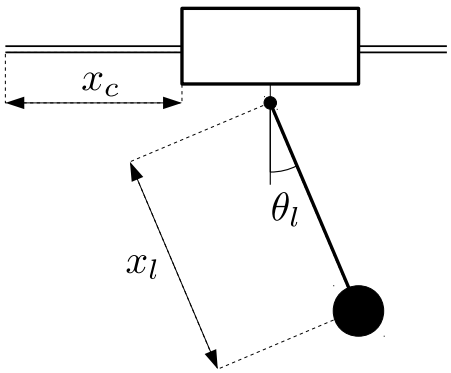


Fig. 4.7 A schematic drawing of the gantry crane setup.

4.4 Experimental setup and simulation results

4.4.1 Experimental setup

Experimental setup represents a closed-loop system that consists of a gantry crane model and a heterogeneous computer running the predictive controller. The closed-loop simulation is managed by Protoip, which was introduced in Section 3.2. Xilinx Zynq-7000 XC7Z020 system-on-a-chip with dual-core ARM Cortex-A9 processor (only one core was used in this work) and Artix-7 FPGA logic was used as a target platform.

Closed-loop simulations were performed in the loop with a Simulink Simscape library-based [62] gantry crane model, which was provided by MathWorks. Using Simscape library allowed incorporation of various types of friction including viscous, breakaway and Coulomb frictions. The setup consists of a payload (m = 0.47 kg) hanging on a rope attached to a moving cart (Figure 4.7). The system is actuated by two Pulse Width Modulation (PWM) controlled motors, which move the cart along the rail and lift/lower the load. Motors are equipped with internal velocity controllers, hence the inputs to the system are velocity setpoints. Cart position x_c, rope length x_l and deflection angle θ_l are measured by encoders with a resolution equal to 4096 pulses per rotation. Corresponding velocities are estimated using a second order low-pass filter cascaded with a differentiator with Laplace transform

$$Y_{est}(s) = \frac{s\omega^2}{s^2 + 2\omega\zeta + \omega^2}. \tag{4.7}$$

In this experiment $\omega = 20\pi$ rad/s, $\zeta = 0.7$. For the purpose of digital implementation (4.7) is discretized using a Tustin transformation with sampling time $T_{est} = 0.01$s. Note that estimator and controller can be sampled at different rates.

The motion of the crane with a variable cable length can be described by [30]

$$\ddot{\theta}_l x_l + 2\dot{x}_l\dot{\theta}_l + \ddot{x}_c\cos(\theta_l) + g\sin(\theta_l) = 0, \tag{4.8}$$

where g is the acceleration of gravity.

It is assumed that the dynamics between the motor velocity setpoints and corresponding velocities are described by the first-order lag relation with Laplace transform

$$Y_{motor}(s) = \frac{1}{\tau s + 1}, \tag{4.9}$$

where τ is the time constant. The following state-space model can be obtained:

$$\dot{x}_c = v_c \tag{4.10a}$$

$$\dot{v}_c = \frac{-v_c + u_c}{\tau_c} \tag{4.10b}$$

$$\dot{x}_l = v_l \tag{4.10c}$$

$$\dot{v}_l = \frac{-v_l + u_l}{\tau_l} \tag{4.10d}$$

$$\dot{\theta}_l = \dot{\theta}_l \tag{4.10e}$$

$$\ddot{\theta}_l = \frac{1}{x_l}\left(\frac{-v_c + u_c}{\tau_c}\cos(\theta_l) + g\sin(\theta_l) + 2v_l\dot{\theta}_l\right) \tag{4.10f}$$

where v_c and v_l denote corresponding velocities. Time constants $\tau_c = 0.13s$ and $\tau_l = 0.07s$ were identified using sum-of-sinusoids input signal postprocessing data with the nonlinear least squares algorithm from [62].

In this work controller performance was initially tested by using the same model for predictive control and plant simulation adopting the CasADi [3] front-end of the CVODES numerical integrator [40]. Following the initial assessment the heterogeneous MPC controller was tested in the loop with Simscape model within the Simulink environment. Incorporating the Zynq platform into the Simulink environment was performed using the Protoip Application Programming Interface (API),

which is implemented in C, and the Simulink Coder package, which allows calling external C/C++ functions.

4.4.2 Simulation results

Four implementations (software only and three heterogeneous) of Algorithm 4 were deployed and tested with Protoip. Single precision floating point data arithmetic was used, clocking the CPU and FPGA at 667 MHz and 167 MHz, respectively. The following control objectives were selected:

$$h(u,x,s) = \begin{pmatrix} x_c + x_l \sin(\theta_l) \\ x_l \cos(\theta_l) - 0.5 \\ \dot{\theta}_l \\ 10^{-4} u_c \\ 10^{-4} u_l \end{pmatrix}, \quad h_T(x_N, s_T) = \begin{pmatrix} x_c + x_l \sin(\theta_l) \\ x_l \cos(\theta_l) - 0.5 \\ \dot{\theta}_l \end{pmatrix}. \tag{4.11}$$

Note that this formulation has proven to be a tuning-free solution for gantry crane control, in contrast to the standard quadratic objective approach, which often relies on a time consuming tuning process [28]. Actuation limits dictated by the motor characteristics were incorporated as input constraints:

$$-0.15 \leq u_c \leq 0.15 \tag{4.12a}$$

$$-0.15 \leq u_l \leq 0.15 \tag{4.12b}$$

For all designs considered $T_s = 100$ ms, the number of interior-point iterations $n_{iter} = 15$ and the number of MINRES iterations was set to be equal to corresponding linear system size. Suitability of these parameters was experimentally confirmed based on closed-loop simulation results.

With the proposed implementation computation time can be traded off against resource usage either by changing the level of parallelism (i.e. the number of MAC units for matrix-vector multiplication) in the hardware accelerator or by shifting MINRES algorithm layers between software or hardware. These tradeoffs are shown in Figure 4.8. Resource usage is calculated as a Euclidian norm of a vector that contains the relative utilization of different FPGA resources (LUTs, FFs, DSPs and BRAMs). For each heterogeneous implementation (HG_1, HG_2 and HG_3) the parallelism factor P was varied from 1 to N. Note that although $N = 10$, there are only 6 designs for

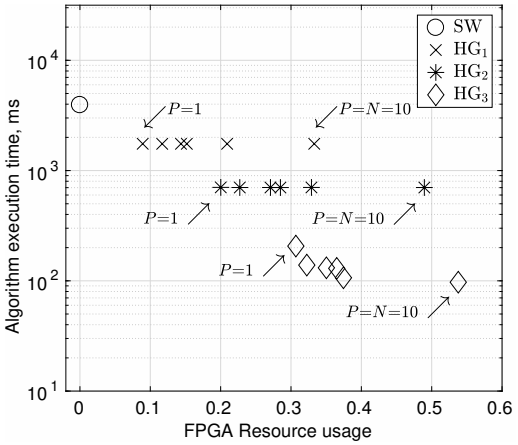


Fig. 4.8 Trading off algorithm execution time against resource usage by splitting the workload between software and hardware. For all implementations $N = 10$.

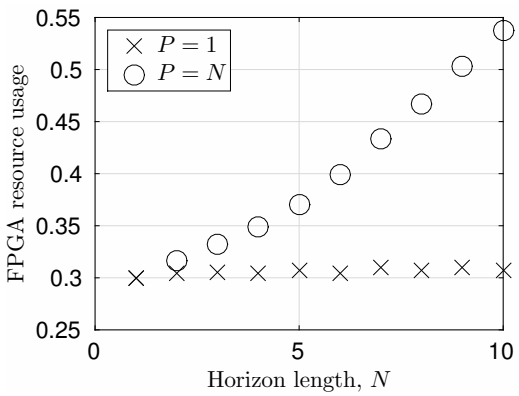


Fig. 4.9 The impact of horizon length on FPGA resource usage with HG_3 implementation.

each implementation. This happens because some values of P are identified as inefficient at code generation stage. For this example any design with $P = 6$ to 9 will lead to the same performance as $P = 5$. We highlight some other conclusions that can be drawn based on Figure 4.8:

- Considering execution time and resource usage as two contradicting design objectives within a multi-objective optimization problem, it can be seen that Pareto-optimal designs are achieved by splitting the workload between software and hardware in different ways. This justifies the flexibility of the software-hardware splitting of the proposed implementation.

- For HG_1 and HG_2 implementations increasing matrix-vector multiplication parallelism does not result in any speed up. This happens due to the fact that the Lanczos kernel (implemented on the FPGA) and the remaining part of the MINRES algorithm (implemented on the CPU) run in parallel and the overall execution time is defined by the slowest branch, which is the remaining part of MINRES.

- For the test case considered the maximum speedup of hardware-accelerated implementations over a pure software realization is 36x. Since this speedup comes at the price of significantly increased resource usage, it might be reasonable to compare the fastest design against other Pareto-optimal designs before making the final design choice.

Increasing the prediction horizon N, while improving closed-loop performance, normally results in implementations with high resource consumption. Consider Figure 4.9 that visualizes scaling of

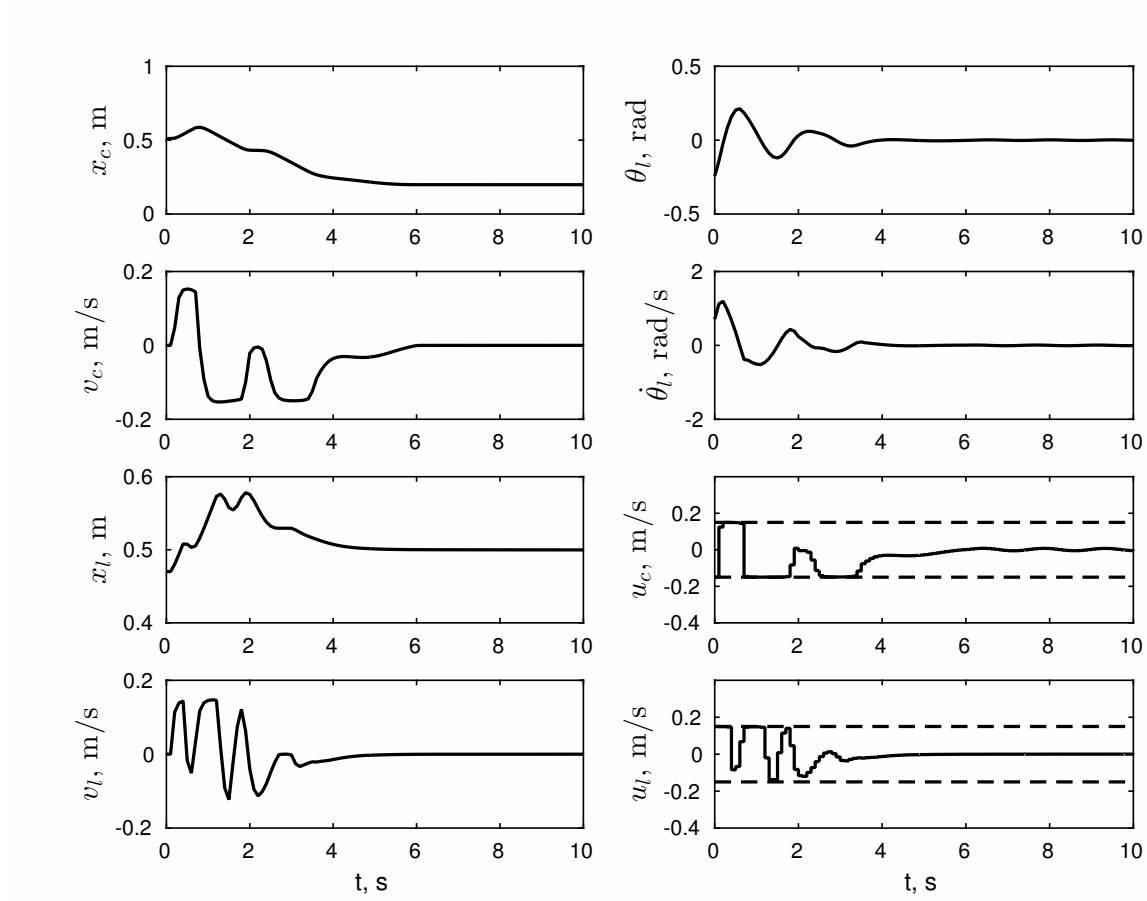


Fig. 4.10 Processor-in-the-loop simulation results with HG_3 implementation. $N = 10$, $P = 10$.

FPGA resource usage with the prediction horizon for two extreme cases: $P = 1$ and $P = N$. It can be observed that changing the degree of parallelism allows taking control over resource scaling. Note that for sequential processing of KKT matrix blocks ($P = 1$) resource scaling can hardly be noticed. This happens due to the fact that the memory footprint is so small that fixed-size FPGA memory blocks are not fully filled for $N = 1$ nor for $N = 10$. Further synthesis details can be found in B.2.

Finally we present closed-loop simulation results obtained with a HG_3 implementation in the loop with Simscape model (Figure 4.10). Starting from the initial condition $\hat{x} = [0.5\ 0\ 0.7\ 0\ -0.2\ -0.5\ -0.15\ -0.15]^T$ the controller drives the system to the desired point according to the objective (4.11) while satisfying constraints (4.12).

4.5 Summary and future work

This chapter presented a heterogeneous implementation of an interior-point-based nonlinear predictive controller and experimental validation of the controller in the loop with a gantry crane model. The following conclusions can be drawn out of this work:

- Accelerating the linear algebra routines in hardware within a heterogeneous computer implementation can result in a significant speedup over a software-only implementation.
- Performance of a heterogeneous computer-based implementation can be efficiently traded off against resource usage by shifting the computational workload between the CPU and the FPGA, while varying the amount of parallelism for a given part of an algorithm might be less efficient or even completely gainless.
- Offline scheduling for sparse matrix vector multiplication allows avoiding data dependencies and hence building efficient data pipelines, which result in faster implementations.

Further work could be focused on automating the design process by formulating the NMPC design problem as multi-objective optimization problem in order to identify design trade offs in a systematic way as in [50]. The list of possible design variables can include both hardware parameters (e.g. parallelization level) and software parameters (e.g. horizon length or number of iterations) parameters.

Chapter 5

Automatic Software and Computing Hardware Co-design for Predictive Control

This chapter presents an application of a bi-objective optimization algorithm to predictive controller design. The proposed approach allows automation of the selection of design parameters while achieving an optimal trade off between controller performance and computational resource usage.

The chapter is organised as follows. Following the introduction in Section 5.1, an approach for formulating predictive controller design as an optimization problem is presented in Section 5.2. Possible ways of formalising design objectives and constraints are discussed within the section. Following that, Section 5.3 reviews existing algorithms for solving the resulting optimization problem and justifies using the BiMADS algorithm [8] for solving MPC design problems. Two case studies are considered in Section 5.4: design of CPU-based and FPGA-based implementations of a fast gradient-based predictive controller. Section 5.5 concludes the chapter.

5.1 Introduction and related work

Model predictive controller design is a multidisciplinary problem that involves tuning several coupled design parameters. Traditionally MPC controllers were tuned manually, with a trial and error approach, which cannot be considered as a viable option for most present-day applications, considering the number of design parameters and design evaluation time [46]. Moreover, manual tuning

often requires understanding the nature of the controlled dynamical system and MPC controller with the underlying optimization solver. Available tuning guidelines for model predictive control, including heuristic and systematic (but not automatic) approaches, are reviewed in [32]. Note that only high level optimal control problem parameters (e.g. horizon length, weights on states/inputs) are considered in the review paper, without regard to solving the underlying optimization problem.

The full design exploration approach, which can be considered as the simplest way of design automation, leads to unacceptable exponential complexity scaling with respect to the number of parameters. Statistical exploration methods, e.g. Monte Carlo methods [53] and Latin Hypercube Sampling (LHS) [95] attempt to accelerate exploration by randomising the sampling process. However, all above techniques *explore* the design space without *exploiting* the knowledge about evaluated designs. As a result, statistical algorithms often achieve uniform distribution in the parameters space, without giving priority to the most promising (in terms of performance criteria) areas and hence waste time evaluating unpromising implementations. However, Monte Carlo methods and LHS can be used for identifying an initial guess for other algorithms.

An alternative approach for taking humans out of the design loop is based on systematic optimization [82]. The following applications of optimization techniques to MPC design can be found in the literature:

- [94] presents an application of multi-objective optimization to the Van de Vusse reaction considering two contradicting objectives: maximizing the desired product and minimizing the unwanted product. Several multi-objective optimization methods are employed in this work: normalized normal constraint [63], normal boundary intersection [26] and weighted sum.

- Study [91] compares systematic multi-objective optimization-based parameter tuning using enhanced normalized normal constraint method [79] against Monte Carlo simulations. The case study considers NMPC applied to a biochemical system.

- Another algorithm for tuning NMPC controllers with application to chemical processes is presented in [59]. Instead of exploring the trade-offs the algorithm attempts to find a single compromise design using the approach [106].

Other examples of automatic tuning of MPC controllers can be found in [104] and [105]. The studies outlined above share certain common features:

- Multi-objective nature of MPC design problem is acknowledged and hence dedicated multi-objective optimization algorithms are used.
- Contradicting control objectives are considered as design goals without explicit optimization of computational complexity.
- Optimization solver parameters are not systematically tuned, it is assumed that optimal control problems are solved to optimality. This might be a valid assumption for slow dynamical systems, e.g. chemical reactors, but not for fast applications, e.g. robotics.
- Underlying computational platform parameters are not involved in the tuning process.

In contrast, in this work we automate MPC controller design considering computational resources as a design objective. The control performance can be traded off against resource usage by tuning both algorithmic and computational hardware-related parameters. The next section will present a formal definition of design objectives, constraints and parameters.

5.2 Formulating the design problem as an optimization problem

5.2.1 Design objectives and constraints

Performance

Quantifying controller performance is not a trivial task: depending on the nature of the dynamic system and design requirements several performance criteria might be considered. Fortunately, optimization-based controllers often perform explicit performance optimization and hence can be evaluated using objective measures. Depending on the application different control objectives might be selected, e.g. energy consumption or settling time. It should be emphasized that the solution of a single optimal control problem cannot serve as a performance indicator, since the ultimate goal is achieving desired closed-loop behaviour [48]. Instead, a closed-loop cost function should be calculated based on a closed-loop simulation.

Considering the closed-loop cost function as an objective within an optimization framework, it is important to take into consideration the continuity and monotonicity properties of the cost function. According to [11], even for a constrained LQR formulation, which is arguably the simplest MPC setup, neither continuity nor monotonicity with respect to horizon length and sampling time

can be guaranteed in general. This significantly limits the range of optimization tools that can be used for MPC design optimization.

Computation time

In relation to CPU implementations, where several algorithms might share the same hardware platform, algorithm execution time becomes both a design objective and design constraint. On the one hand, minimizing algorithm execution time keeps processor load low and hence enables sharing processor time with other algorithms. On the other hand there is a fundamental constraint for MPC design problems: in order to implement the controller in real-time, the optimization algorithm execution time has to be smaller than the sampling time of the system. There are certain exceptions to this rule [19], which are not considered in this work.

In contrast, for FPGA implementations, where a circuit is synthesised for one particular algorithm, minimizing computation time would not give any benefits, since the logic cannot be reused by other algorithms. Moreover, for certain cases it could be worthwhile to increase computation time by decreasing the circuit size and hence saving resources. However, computation time can only be increased up to the sampling time, which can be formalized as a design constraint.

Note that, for a given algorithm the CPU implementation execution time would not be fully predictable because of a complex memory hierarchy. For FPGA circuits, execution time (in terms of clock cycles) can often be determined based on the architecture and hence efficiently predicted before circuit synthesis.

Computational logic usage and energy consumption

An FPGA designer often aims to minimize the amount of silicon that is required for implementing a particular control algorithm to get a size- and cost-efficient solution. As discussed in Section 2.1.5, a modern FPGA has the following resources: flip-flops, lookup tables, block RAMs and DSPs. We measure the silicon usage (or computational logic usage) as follows:

$$R_{FPGA} = \sqrt{R_{FF}^2 + R_{LUT}^2 + R_{DSP}^2 + R_{BRAM}^2}, \tag{5.1}$$

where R_{FF}, R_{LUT}, R_{DSP} and R_{BRAM} denote relative utilization of each resource type. Euclidean norm can be considered as a compromise between the L^1 and L^∞ norms, where the former would

not take into account a possible imbalance between different types of resources and the latter would penalize only the most heavily used resource.

In contrast to taking the average (which in this case would be equivalent to the L^1 norm), the Euclidean norm automatically accounts for the fact that a certain implementation is constrained by the most heavily-used resource.

There is seldom a linear correspondence between circuit size and energy consumption. For instance, in some cases it could be energy-efficient to create a large circuit by parallelizing all computations in order to quickly perform all calculations and switch to idle mode [48]. In such cases, energy consumption may be considered as a separate design objective, which is particularly important for energy-autonomous embedded platforms.

In this work we use the vendor's synthesis tool reports to get computational logic usage information and energy consumption estimation, i.e. the impact of design parameters on design objectives is modelled as a black box. An alternative approach would be based on building an FPGA resource usage model and querying the model instead of synthesizing a circuit for every design evaluation. Since FPGA circuit is not synthesized in this case, processor-in-the-loop simulations would be replaced by software-in-the-loop simulations with software emulation of number representation. As a result this alternative approach can potentially save design evaluation time, since time-consuming circuit synthesis process would be replaced by a model query. However, building the resource usage model might require deep understanding of the synthesis process. Moreover, this model would be vendor-specific, which potentially limits its application scope. Furthermore, in certain cases software-in-the-loop emulation might slow down evaluating the closed-loop performance, since the target FPGA arithmetic might be different from the native arithmetic of a standard desktop machine.

5.2.2 Design parameters

Horizon length and sampling time

Horizon length and sampling time are fundamental design variables that have a crucial impact both on closed-loop performance of a system and computational hardware requirements. These two parameters, being tightly coupled with each other, define the quantity of predicted information in a predictive controller. Horizon length defines the "vision distance", while sampling rate sets the

"quality of the picture" [11]. Sampling frequency also defines the response delay of the controller. An overview of techniques for manual tuning of these parameters can be found in [32].

Note that MPC problem sampling time might differ from the control signal update period. However, in this work we set control signal update period equal to MPC problem sampling time. For the the closed-loop simulation we assume that the controller delay is equal to zero, i.e. control input is applied immediately after receiving the current state information.

Objective function weights

Although the idea of explicit performance optimization is the main advantage of MPC, in practice the original *economic* objective is often replaced/augmented with a quadratic function. The main reasons behind this approach are:

- Regularization. For example [107] proposes a time-optimal MPC formulation, which is based on augmentation of the original objective with a quadratic term.
- Lack of computational resources. Many of existing real-time optimization solvers are tailored for QPs, which makes it impossible to handle nonlinear objective functions directly.

Tuning the quadratic function weight parameters might have a significant impact on the controller performance and computational complexity. In this work the state penalty matrix is treated as a design variable, while the input penalty is considered as a fixed parameter.

Problem formulation: condensed vs non-condensed

Eliminating the states from decision variables by expressing them via the current state and input sequence leads to a compact condensed formulation [43], which results in a worst-case cubic scaling of the number of floating point operations in horizon length for the primal-dual interior point algorithm iteration. A sparse formulation implies the opposite approach: keeping the dynamics in constraints and treating both inputs and states as decision variables. Although the problem size becomes larger, exploiting the sparsity pattern allows linear scaling of computational effort in horizon length. The condensed and non-condensed formulations can be considered as two extreme points of a *sparsity level* design variable. Controlling the level of sparsity can be achieved by dividing the prediction horizon into sub-intervals and performing partial condensing. This provides

the possibility of adjusting the block size of linear algebra problems in order to find the optimal level of sparsity in terms software and hardware resources utilization [9].

Optimization algorithm parameters

Optimization algorithms that solve optimal control problems numerically often have many tuning parameters on the underlying levels. The first fundamental design choice is the algorithm type [48]:

- first vs second order methods
- interior point vs active set algorithms
- gradient-based vs derivative-free approaches

In addition to making high level design decisions, it is important to tune low level parameters, which might include:

- number of iterations / termination criterion
- barrier parameter update strategy for interior-point algorithms
- globalization strategy (line search vs trust region)

The above parameters must be tuned with respect to closed-loop performance, which is not necessary correlated with a single optimization problem's optimality conditions [48]. For example, a sub-optimal controller with a longer prediction horizon might perform better than an optimal controller with a shorter horizon. More details on sub-optimal MPC can be found in [80].

Number representation

Different types of number representation were discussed in Section 2.1.5. There are usually two types of choices that have to be made:

- The conceptual choice of data representation type: e.g. floating point or fixed point.
- The numbers of bits to be allocated for different parts of a number, e.g. for mantissa and exponent in floating point arithmetic.

Data-level parallelism and pipelining

Data-level parallelism and pipelining were discussed in Section 2.1.5. The main algorithmic choices in relation to these techniques are:

- The number of parallel processors.
- The number of pipeline stages, i.e. *pipeline depth* (refer to Figure 2.4 on page 41).

It should be emphasized that parallelism and pipelining affect algorithm execution time and resource usage, which may or may not have an impact on the closed-loop performance.

5.2.3 The resulting optimization problem

The design parameters considered in Section 5.2.2 can be classified in two categories: software parameters (e.g. prediction horizon length) and hardware parameters (e.g. number representation). Conventional approaches propose sequential design: initially the software algorithm is designed at a high level of abstraction without regard to the intended hardware platform and, following this, the algorithm is implemented on a hardware platform by selecting appropriate hardware parameters. This decoupled approach usually leads to inefficient utilization of available resources [48]. In contrast, the co-design approach implies simultaneous design of both software and hardware components in order to achieve the best possible performance for a given set of available resources. However, improvement of the closed-loop performance cannot be considered as the only design objective. As can be seen from Section 5.2.1 there are often several contradicting design objectives, which might include performance and computational hardware resource usage. Instead of looking for one optimal design (which often does not exist due to conflicts between objectives), engineers might make a decision based on the whole series of Pareto optimal designs, i.e. designs that cannot be improved in terms of one objective without worsening at least one of the other objectives.

The problem of investigating design trade-offs is usually formalized as a multi-objective optimization (MOO) problem. The main bottleneck that prevents efficient solution of MOO design problems in relation to MPC are the properties of the objective functions:

- Long function evaluation time. Evaluation of the design objective functions requires time-consuming simulations to be performed.

- Absence of derivative information. There are no analytical expressions for accurate estimation of the design objectives derivatives.
- Mixed domain. Design variables can be both discrete (e.g. number of bits for data representation) and continuous (e.g. sampling rate).
- Noisiness. For example, the same HLS code may result in different resource usage values depending on a vendor's software version or other factors that are not taken into account by conventional models.

The next section will review existing algorithms for solving multi-objective optimization problems with focus on BiMADs, a bi-objective version of a mesh adaptive direct search algorithm.

5.3 Derivative-free multi-objective optimization

5.3.1 Problem statement

We consider the following multi-objective optimization problem:

$$\min_{p} \quad f(p) \coloneqq \begin{pmatrix} f^{(1)}(p) \\ \vdots \\ f^{(l)}(p) \end{pmatrix} \tag{5.2a}$$

$$\text{subject to} \quad p \in S, \tag{5.2b}$$

where S is q-dimensional decision space. Since the objectives of MOO are often contradicting, there is no single solution to the problem. Instead, a set of *Pareto optimal* solutions can be obtained.

Definition 5.3.1. *Pareto optimality. A feasible solution $p^* \in S$ is Pareto optimal if there does not exist another feasible solution $p \in S$ such that $f_i(p) \leq f_i(p^*)$ for all $i \in \{1,\dots,l\}$ and $f_i(p) < f_i(p^*)$ for at least one index $j \in \{1,\dots,l\}$.*

The Pareto frontier is the set of all existing Pareto optimal points.

Definition 5.3.2. *Pareto dominance. A point $y' = f(p')$ is said to (strongly) dominate $y'' = f(p'')$ iff $\forall i \in \{1,\dots,l\} : y'_i \leq y''_i$ and $y' \neq y''$. The shorthand notation for this is $y' \prec y''$. Analogously, for weak dominance, $y' \preceq y''$ means $\forall i \in \{1,\dots,l\} : y'_i \leq y''_i$.*

$P(U)$ is the set of non-dominated points for a given set of evaluated points U, i.e. the current approximation of the Pareto frontier. The quality of a Pareto frontier approximation can be assessed by means of the *hypervolume*.

Definition 5.3.3. *Hypervolume. For a given reference point y_{ref} and Pareto frontier approximation P, the hypervolume is defined as a set of points in the objective space $\{y \preceq y^{ref} \in \mathbb{R}^l | \exists y' \in P : y' \preceq y\}$.*

The Lebesgue measure of the hypervolume $L(P, y_{ref})$, i.e. hypervolume space, defines the quality of the Pareto frontier approximation.

5.3.2 Review of algorithms for derivative-free multi-objective optimization

Derivative-free single-objective optimization

Deterministic algorithms for single-objective derivative-free optimization can be classified into [21]:

- *Trust-region interpolation algorithms*. Trust-region algorithms propose building a local approximation of the objective function based on evaluated samples. Based on this approximation, the function is minimized inside the trust region.
- *Line search algorithms* for derivative-free optimization are conceptually similar to their derivative-based counterparts: they perform search along a particular direction. However, for derivative-free algorithms, the search direction is calculated without gradient information.
- *Direct Search Methods* (DSMs) do not attempt to approximate derivatives either explicitly or implicitly. Instead, optimization is based on evaluation of a finite set of points around the current solution guess, so that a point with smaller objective function value can be found.

Among the considered classes of algorithms, only direct search methods of directional type have been extensively studied in relation to multi-objective optimization [24], hence rest of the section will be focused on DSMs.

Each iteration of a DSM is split into two parts: a *search step* and *poll step*. The latter step must be rigidly defined to guarantee convergence, while the former is more flexible and can be tuned to improve numerical performance. Assuming there is a given feasible solution guess, the general structure of a DSM of directional type can be described as follows [21]:

1. **Search step**. Evaluate the objective function at a finite set of points. If any of these points provide a better objective value compared to the current guess, declare the iteration as successful and skip the poll step.

2. **Poll step**. Perform a local search around the current best point by evaluating a set of poll points, which are defined by poll directions and a step size. Depending on the result of this search, declare the iteration as successful or unsuccessful.

3. **Mesh parameter update**. Reduce step size for unsuccessful iterations, increase or maintain step size for successful ones.

Practical examples of DSM algorithms of directional type are the coordinate-search method [21] and the Mesh Adaptive Direct Search (MADS) algorithm [6].

Derivative-free multi-objective optimization

There are two fundamentally different ways for tackling multi-objective optimization problems:

- *Direct* algorithms for multi-objective optimization attempt to approximate the Pareto frontier directly.

- *Scalarization-based* approaches propose converting the multi-objective problem into a sequence of single-objective problems.

The Direct Multisearch (DMS) algorithm [25], which belongs to the class of direct multi-objective optimization algorithms, is a generalization of DSM algorithms for single-objective optimization. The main components of a DMS are the same as DSM as discussed in the previous subsection: the search step, the poll step and mesh update. The algorithm keeps a list of non-dominated points, which represents the current Pareto frontier approximation. The local poll search is performed around several non-dominated points. Successfulness of an iteration is decided based on the changes in the Pareto frontier approximation. The search step, similarly to single-objective DSM algorithms, is flexible and is not required for convergence [24]. However, it can help to improve the distribution of the points along the Pareto frontier, although this is not a systematic way of ensuring uniformity of the frontier.

A classical example of scalarization-based algorithms is the weighted-sum method. The method scalarizes the vector of objectives into a single objective by taking an affine combination of all

the objectives. Tuning the weights of the scalarized function allows one to move along the Pareto frontier. However, in this case movement along the frontier happens in a non-systematic way. As a result, some regions become over-represented and others may suffer from lack of information. In addition, this approach cannot deal with non-convexities and discontinuities in the Pareto frontier and hence loses some Pareto optimal solutions.

The BiMADS algorithm [8] performs scalarization in a different way. For a problem with two objectives the scalarized function is given by

$$\phi_r(p) = \begin{cases} -\prod_{i=1}^{2}(r_i - f^{(i)}(p))^2 & \text{if } f(p) \preceq r, \\ \sum_{i=1}^{2}(\max\{0,(f^{(i)}(p) - r_i)\})^2 & \text{otherwise,} \end{cases} \tag{5.3}$$

where r is a reference point in the objective space. The problem of minimizing $\phi_r(\cdot)$ is solved by the MADS [6] algorithm. The formulation (5.3) with appropriate selection of a reference point allows the recovery of all Pareto optimal solutions, which is not the case for the weighted-sum reformulation. For achieving good distribution of non-dominated solutions, the reference point must be selected in a particular way. The BiMADS algorithm proposes a formal method for identifying the biggest gap in the Pareto frontier approximation and presents a procedure for selecting an appropriate reference point. Details can be found in [8].

In this work, the BiMADS algorithm will be used for solving the multi-objective co-design problem. The main advantages of the algorithm are:

- Both BiMADS and the underlying single-objective MADS solver are supplied with a mathematical proof of convergence [8, 6], which is not the case for other considered algorithms.
- BiMADS has proven to be efficient for solving real-world engineering problems [33].
- There is a free software application that implements the algorithm, namely NOMAD [55]
- NOMAD provides various interfaces, including a Matlab front-end, which simplifies integration with Protoip (Section 3.2).

The application scope of BiMADS is limited to bi-objective problems, which can be considered as the main drawback of the algorithm.

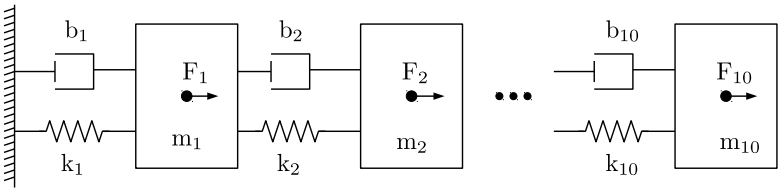


Fig. 5.1 Mass-spring-damper system.

5.4 Case studies

5.4.1 Fast gradient-based controller for a mass-spring-damper system - CPU-only implementation

The system under control represents a chain of ten masses that are connected via springs and dampers (Figure 5.1). The first mass is also connected to a fixed wall. Each mass can be actuated with an input force that has input and output limits. It is assumed there is no gravitational force. The system can be modelled with a continuous-time linear state space model:

$$\dot{x}(t) = A_c x(t) + B_c u(t) \tag{5.4}$$

where $A_c \in \mathbb{R}^{n\times n}$, $B_c \in \mathbb{R}^{n\times m}$ are continuous-time state and input matrices. For a system of ten masses $n = 20$ and $m = 10$.

The following optimal control formulation is considered:

$$\underset{u,x}{\text{minimize}} \quad \int_0^T \left(\frac{1}{2}x^T(t)Q_c x(t) + \frac{1}{2}u^T(t)R_c u(t) + x^T(t)W_c u(t)\right)dt + \frac{1}{2}x^T(T)P_c x(T) \tag{5.5a}$$

$$\text{subject to} \quad x(0) = \hat{x} \tag{5.5b}$$

$$\dot{x}(t) = A_c x(t) + B_c u(t), \quad \forall t \in [0,T] \tag{5.5c}$$

$$u_{min} \leq u(t) \leq u_{max}, \quad \forall t \in [0,T] \tag{5.5d}$$

where $Q_c \in \mathbb{S}^n_+$, $R_c \in \mathbb{S}^m_{++}$, $W_c \in \mathbb{R}^{n\times m}$ and $P_d \in \mathbb{S}^n_{++}$ are state, input, cross and terminal penalty matrices accordingly. $\mathbb{S}^n_{++}(\mathbb{S}^n_+)$ denotes a set of positive (semi-)definite matrices.

In this experiment the following prediction matrices were used:

$$R_c = I_{m\times m} \otimes [0.0001], \quad Q_c = I_{n\times n} \otimes \begin{bmatrix} 1 & 0 \\ 0 & q_{speed} \end{bmatrix}, \quad W_c = 0_{m\times n}, \quad P_c = Q_c, \tag{5.6}$$

where I and $0_{m\times n}$ denote identity and zeros matrices accordingly. Tuning q_{speed}, which is a design parameter, allows the changing the ratio between penalising masses positions and velocities.

For the purpose of digital control, the continuous-time state-space model (5.4) and optimal control problem (5.5) are discretized, assuming a zero-order hold:

$$\underset{u_0\dots u_{N-1},x_0\dots x_N}{\text{minimize}} \quad \sum_{k=0}^{N-1}\left(\frac{1}{2}x_k^T Q_d x_k + \frac{1}{2}u_k^T R_d u_k + x_k^T W_d u_k\right) + \frac{1}{2}x_N^T P_d x_N \tag{5.7a}$$

$$\text{subject to} \quad x_0 = \hat{x} \tag{5.7b}$$

$$x_{k+1} = A_d x_k + B_d u_k, \quad \text{for } k = 0,1,\dots,N-1 \tag{5.7c}$$

$$u_{min} \le u_k \le u_{max}, \quad \text{for } k = 0,1,\dots,N-1 \tag{5.7d}$$

Note that the discrete-time penalty matrices Q_d, R_d, W_d and P_d are model-dependent [11].

The optimal control problem (5.7) can be transformed into a quadratic programming problem by eliminating the states, which leads to the condensed formulation

$$\text{minimize} \quad \frac{1}{2}\theta^T H\theta + \theta^T h \tag{5.8a}$$

$$\text{subject to} \quad \theta_{min} \le \theta \le \theta_{max} \tag{5.8b}$$

Note that the gradient term h depends on the current state, while the Hessian H is fixed and hence can be precalculated offline. More details on condensed and sparse formulations for predictive control can be found in [43]. Since (5.8b) has the form of box constraints, calculating projection on the feasible set becomes computationally cheap. This facilitates using Nesterov's projected gradient algorithm [65], also known as the Fast Gradient Method (FGM). The method proposes moving in the anti-gradient direction and performing projection $P(\cdot)$ on the feasible set after each iteration (Algorithm 5). The extra momentum step with a parameter β achieves an optimal convergence rate. The constant step scheme [65] implies $\beta = \frac{\sqrt{L}-\sqrt{\mu}}{\sqrt{L}+\sqrt{\mu}}$, where L is the largest eigenvalue of the Hessian H and μ is the convexity parameter, which is equal to minimum eigenvalue of the Hessian.

The algorithm was implemented on an ARM Cortex A9 processor of the Xilinx Zynq-7000 XC7Z020 system-on-a-chip employing only a single processing core. Using the Protoip toolbox allowed for fast assessment of the controller in the loop with the plant model as shown in Figure 5.2. Details on the Zynq platform architecture and closed-loop testing setup can be found Sections 2.1.6

Algorithm 5 Projected fast gradient algorithm for constrained optimization with constant step size.

1: Initial guess:θ_0
2: $v_0 = \theta_0$
3: **for** $i = 0$ to N_{FGM} **do**
4: $\quad \theta_{i+1} = (I - (1/L)H)v_i - (1/L)h$ ▷ anti-gradient step
5: $\quad z_{i+1} = P(\theta_{i+1})$ ▷ projection on the feasible set
6: $\quad v_{i+1} = (1+\beta)z_{i+1} - \beta z_i$ ▷ extra-momentum step
7: **end for**

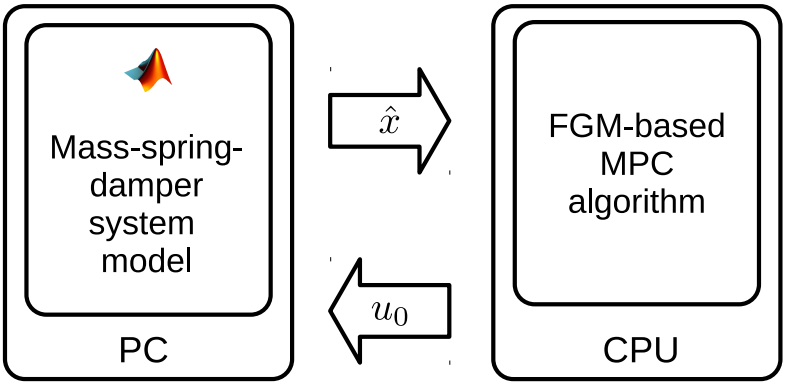


Fig. 5.2 CPU in the loop test setup.

and 3.2, respectively.

The problem of interest is automatic design of a fast gradient-based controller. The following design objectives are considered:

- *Controller performance* is judged based on settling time. In this experiment, settling time is defined as the time elapsed from the beginning of closed-loop simulation to the time at which $\|x(t)\|_2 \leq \varepsilon$, where $\varepsilon = 0.01$. Several simulations with different initial conditions are performed in order to calculate performance measure as a sum of settling times for different initial conditions; the list of initial conditions are presented in Appendix A. Since the performance criterion and MPC objective are different, it is essential to tune the prediction matrices in order to achieve the desired performance.

- *Algorithm computational time*. As discussed in Section 5.2.1, computational time is the main measure of algorithm complexity for CPU implementations.

Design constraints:

- *Algorithm computational time*, in addition to being a design objective, appears in a constraint function: in order to implement the controller in real-time the algorithm execution time has to be smaller than the sampling time of the system (Section 5.2.1).

- *Stability constraint* captures whether the controller was able to stabilize the system. Unstable response might happen due to short horizon, numerical errors or other reasons.

Note that the former constraint is *quantifiable* while the latter is *non-quantifiable*. A quantifiable constraint is a constraint for which the degree of feasibility and violation can be quantified [7]. In this work, non-quantifiable constraints are handled with the *extreme barrier* approach, which implies setting the objective to infinity for all infeasible points and therefore not allowing infeasible iterations. For quantifiable constraints the *progressive barrier* approach is adopted. Progressive barrier constraint handling allows exploiting knowledge of the violation degree by accepting infeasible iterations. Both extreme and progressive barrier approaches are implemented in NOMAD. More details can be found in [7, 55].

The design parameters are the following:

- Horizon length, N in (5.7); bounds: $1 \leq N \leq 12$.

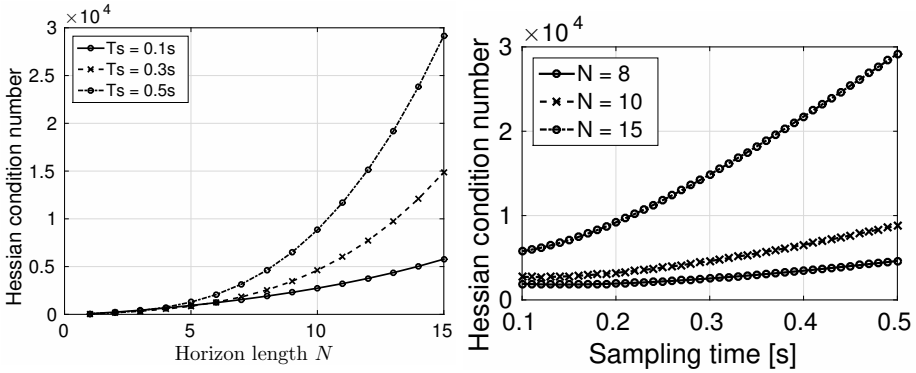
- Sampling time, T_s; bounds: $0.02 \leq T_s \leq 0.5$.

- Number of fast gradient algorithm iterations, N_{FGM} in Algorithm 5; bounds: $20 \leq N_{FGM} \leq 200$.

- State penalty matrix, particularly q_{speed} parameter in (5.6); bounds: $0.2 \leq q_{speed} \leq 5$.

The above parameters are tightly coupled with each other. For example, consider Figure 5.3 that illustrates the impact of horizon length and sampling time on the Hessian condition number. It can be observed that both parameters have a significant impact on the condition number, which in turn affects the convergence rate of the fast gradient algorithm [65]. As a result, the number of fast gradient algorithm iterations N_{FGM} required for convergence will also change. However, N_{FGM} must be selected with respect to closed-loop performance, rather than open-loop optimality conditions, which complicates the tuning process even more.

The above design problem was solved using the BiMADS algorithm. The results are compared to latin hypercube sampling, which is a statistical sampling method commonly used for design



(a) The impact of horizon length.

(b) The impact of sampling time.

Fig. 5.3 The impact of horizon length and sampling time on Hessian condition number.

exploration and for design-of-experiments in particular [95]. For both experiments the number of evaluations was restricted to 200. Design evaluation involves compiling the source code and performing processor-in-the-loop tests, which takes 1-4 minutes for the considered setup, depending on a design complexity and the sampling time. Observe that full design exploration is not a viable approach: even using a coarse grid with ten points for the continuous variables, full exploration will require 217200 evaluations. As can be seen from Figure 5.4, the Pareto frontier returned by BiMADS dominates the front identified by LHS, i.e. any design sampled by LHS is dominated by at least one design identified by BiMADS (see Definition 5.3.2). Moreover, according to the hypervolume profile (Figure 5.5), BiMADS provides a satisfactory approximation of the Pareto frontier on the early stages, which opens the possibility of early termination, depending on the time and/or simulation resource availability.

Consider Table 5.1, which lists designs identified by BiMADS. Designs are sorted according to their settling times in ascending order. It can be observed that most of the design parameters (T_s, N and N_{FGM}) are changing along the Pareto-frontier in a non-monotonic way. This can be explained by the complicated cross-coupling between parameters. As a consequence, manual and heuristic tuning approaches are unlikely to identify these trade offs in an efficient way. The only exception is the weight matrix parameter q_{ratio}, which has the same value for all designs. In this particular case q_{ratio} could be optimized separately from other design parameters.

Given the Pareto frontier (Figure 5.4) a designer will be able to select a particular implementation based on the available processing time. For example:

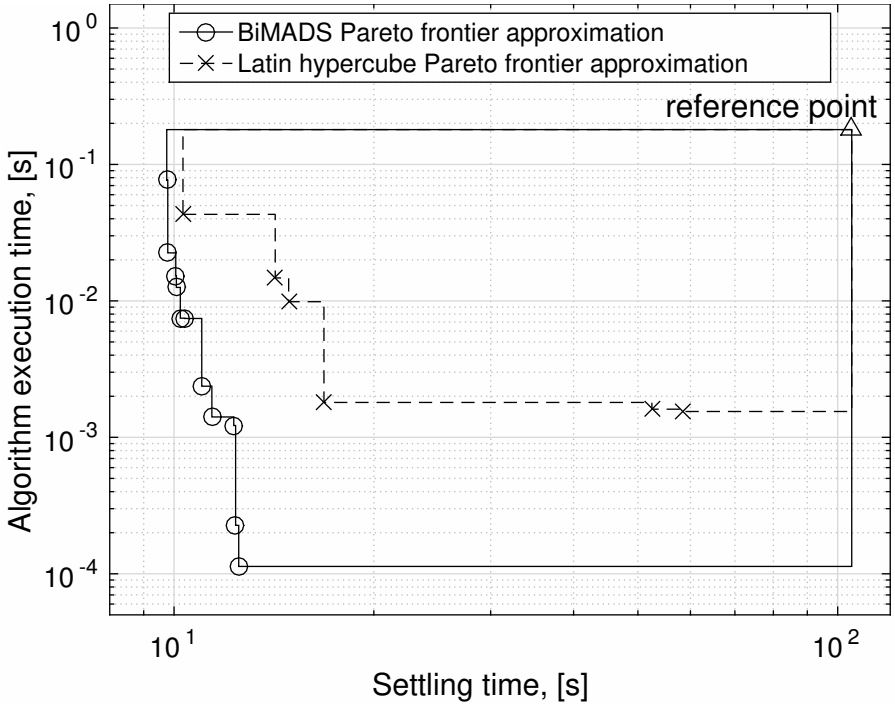


Fig. 5.4 Pareto frontier approximation for CPU implementations of the FGM: BiMADS vs LHS.

Table 5.1 Pareto frontier approximation identified by BiMADS for CPU implementations of FGM (corresponds to Figure 5.4). Designs are sorted according to their settling times in ascending order.

Design parameters				Design objectives	
T_s, s	N	N_{FGM}	q_{ratio}	Settling time, s	Algorithm time, s
0.1880	8	184	0.20	9.7499	0.0769
0.2360	5	136	0.20	9.7845	0.0225
0.3320	4	136	0.20	10.0598	0.0153
0.2360	7	40	0.20	10.0805	0.0126
0.3320	3	112	0.20	10.2197	0.0075
0.3800	3	112	0.20	10.3747	0.0074
0.2840	4	20	0.20	11.0092	0.0024
0.2840	2	40	0.20	11.4023	0.0014
0.4760	1	256	0.20	12.3020	0.0012
0.5000	1	44	0.20	12.3816	0.0002
0.4760	1	20	0.20	12.5158	0.0001

- For the computational budget of 100 ms, LHS-based exploration achieves a settling time of 10.31 s, while optimization-based design allows settling of the plant in 9.75 s.
- Tightening the computational budget to 2 ms leads to sampling times of 16.82 s and 11.40 s for LHS- and optimization-based approaches accordingly.

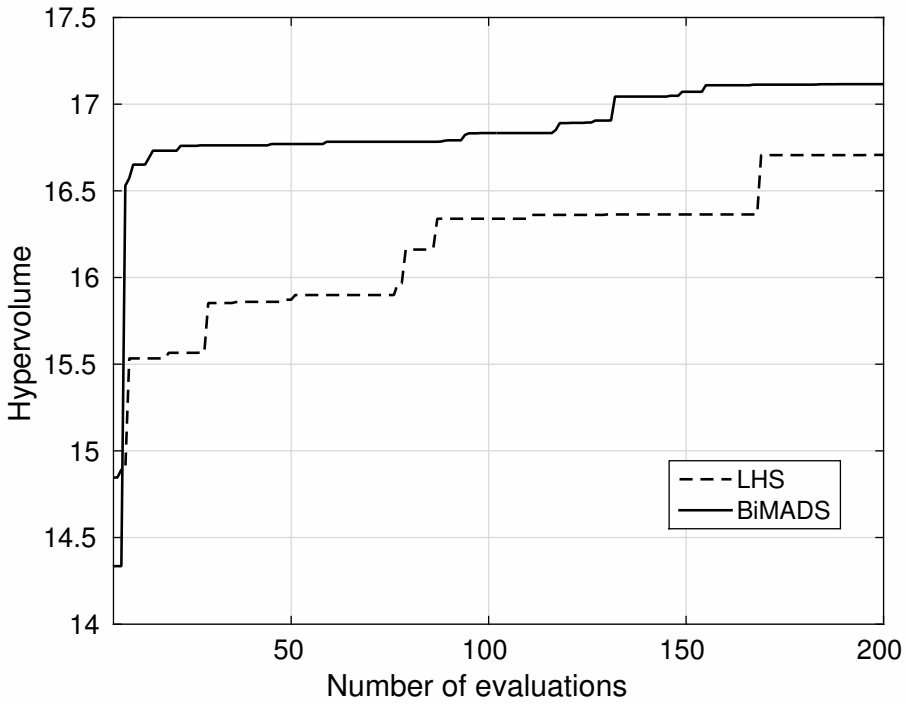


Fig. 5.5 Hypervolume profiles for CPU implementations of the FGM: BiMADS vs LHS.

5.4.2 Fast gradient-based controller for a mass-spring-damper system - FPGA implementation

This case study considers implementation of the Algorithm 5 with fixed point arithmetic on the FPGA logic of Xilinx Zynq-7000 XC7Z020 SoC. The testing setup is similar to that of Section 5.4.1, both in terms of the optimal control problem formulation (5.5) and the plant model (Figure 5.1). The algorithm was implemented with the Vivado HLS FPGA synthesis tool using Protoip for automatic deployment and testing in the loop with the plant model (Figure 5.6). As can be seen

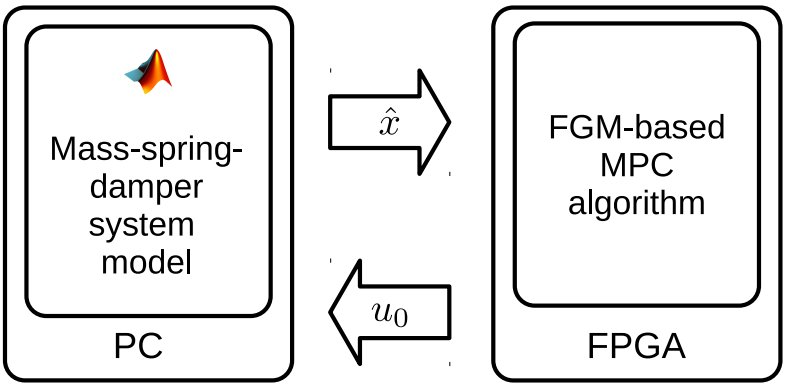


Fig. 5.6 FPGA in the loop test setup.

from Algorithm 5, FGM relies only on addition and multiplication operators, while all divisions can be precalculated offline. The following techniques were used to accelerate vector-vector and matrix-vector operations (lines 4-7):

- *Loop pipelining* [100]. Data pipelining, as a general acceleration technique, was discussed in Section 2.1.5. In relation to loops, pipelining implies overlapping iterations, i.e. starting a new iteration before finishing the previous.

- *Loop flattening* [100] is transforming nested loops into a single loop with multiple counters. Flattening allows efficient pipelining of nested loops. This technique was applied to matrix-vector multiplication (line 4), where loop nests arise when iterating over matrix rows and columns.

As discussed in Section 2.1.5, fixed point arithmetic often introduces overflow and round off errors. The former issue can be addressed by precalculating the upper bound on the largest absolute value of algorithm iterates using interval arithmetic. Regarding overflow errors, the number of fraction bits has to be sufficiently large to maintain numerical stability of an iterative algorithm. A procedure for precalculating the number of integer and fraction bits for fixed point implementations of a fast-gradient algorithm is presented in [45].

The following objectives are considered in this design optimization experiment:

- *Controller performance* is measured similarly to the previous case study (Section 5.4.1) with $\varepsilon = 0.02$.

- *FPGA logic usage*. As discussed in Section 5.2.1 FPGA designers often aim to minimize the amount of logic used for a particular algorithm. Logic usage is measured as the Euclidean norm of relative utilization of each resource type, see (5.1) in Section 5.2.1.

Design constraints:

- *Algorithm execution time*. Similarly to the previous case study, in order to implement the controller in real-time, the algorithm execution time has to be smaller than the sampling time of the system. This constraint is treated with the progressive barrier approach. Note that for FPGA setup (in contrast to CPU), computational time does not appear as an objective. This is explained by the fact that FPGA logic is synthesized for a particular algorithm and cannot be reused for other applications.

- *Objective function convexity constraint.* Due to assumptions on the weight matrices in formulation (5.5), the Hessian of the objective function (5.8a) is positive definite. However, a fixed point representation of the true Hessian may be non-convex because of truncation errors, which might affect convergence of the fast-gradient algorithm. To avoid non-convex formulations, a positivity constraint on the smallest eigenvalue of the fixed point representation of the Hessian must be set. Although this is a quantifiable constraint, which potentially can be treated with a progressive barrier approach, we will use the extreme barrier method that rejects all infeasible iterations. Since identifying Hessian convexity is significantly faster compared to the full design evaluation, which involves circuits synthesis and closed-loop simulation, rejecting infeasible iterations allows saving design time.

The following design parameters are considered:

- Horizon length, N in (5.7); bounds: $1 \leq N \leq 12$.
- Sampling time, T_s; bounds: $0.02 \leq T_s \leq 0.5$.
- Number of fast gradient algorithm iterations, N_{FGM} in Algorithm 5; bounds: $20 \leq N_{FGM} \leq 200$.
- State penalty matrix, particularly q_{speed} parameter in (5.6); bounds: $0.2 \leq q_{speed} \leq 5$.
- Number of fraction bits for fixed point number representation, N_{frac}; bounds: $5 \leq N_{frac} \leq 25$.

Similarly to the previous case study, the multi-objective optimization problem with the above design objectives and constraints was solved using BiMADS algorithm and results were compared to Latin hypercube sampling allowing 200 evaluations for both algorithms. In contrast to software compilation, FPGA circuit synthesis is a time-consuming process, which leads to the design evaluation time in the range of 20-35 minutes. For this case study full design exploration would require 4561200, assuming a grid of ten points for the continuous parameters.

It can be observed from Figure 5.7 that BiMADS outperforms LHS, i.e. the BiMADS Pareto frontier dominates frontier identified by LHS. However, unlike with the previous case study, LHS outperforms BiMADS in the early stages of design exploration, which can be visualized with the hypervolume profile in Figure 5.8. This might happen due to bad selection of initial guesses for BiMADS. Moreover, LHS, being a statistical method, might occasionally identify Pareto optimal

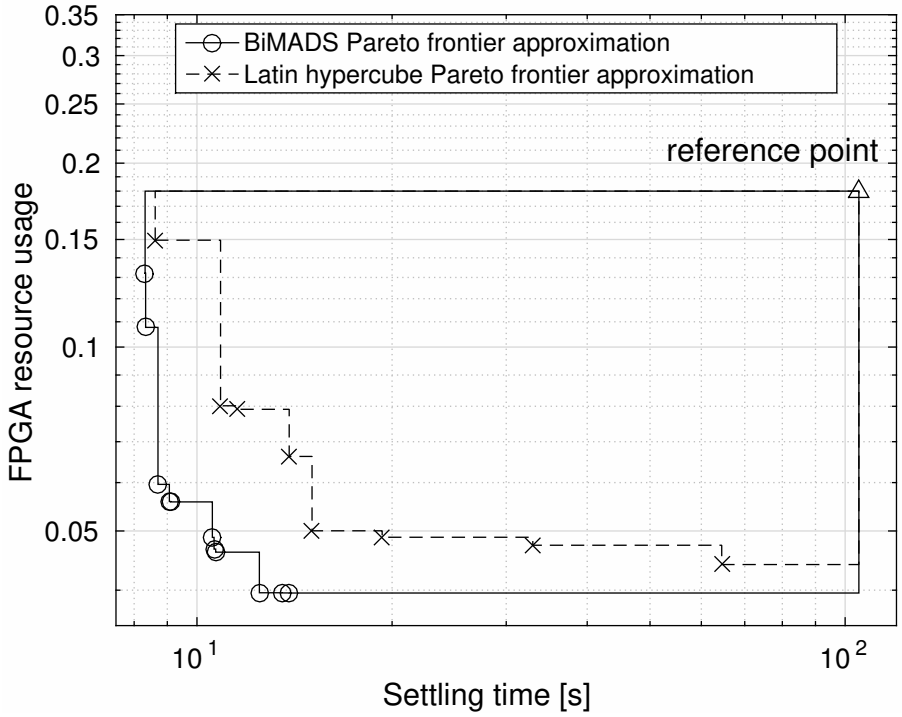


Fig. 5.7 Pareto frontier approximation for FPGA implementations of the FGM: BiMADS vs LHS.

designs faster than deterministic algorithms. It can be observed from Figure 5.8 that, after achieving a certain hypervolume space in the beginning of exploration process, LHS does not improve the Pareto frontier approximation significantly further. In contrast, BiMADS, having a poor initial guess, improves the solution and outperforms LHS when reaching an evaluation limit.

Consider Table 5.2, which lists designs identified by BiMADS. Similarly to the software implementation of the fast gradient algorithm, the weight matrix parameter q_{ratio} is the same for all designs and equal to the lower bound for this parameter. Other design parameters, including the hardware parameter N_{frac}, are changing along the Pareto frontier, trading off resource usage against performance.

Given the Pareto frontier (Figure 5.7) a designer will be able to select a particular implementation based on the available FPGA resources (R_{FPGA}). For example:

- For $R_{FPGA} = 0.1$, LHS-based exploration achieves a settling time of 10.87 s, while optimization-based design allows settling of the plant in 8.71 s.
- Tightening the resource limit to $R_{FPGA} = 0.05$ leads to sampling times of 19.31 s and 10.56 s for LHS- and optimization-based approaches accordingly.

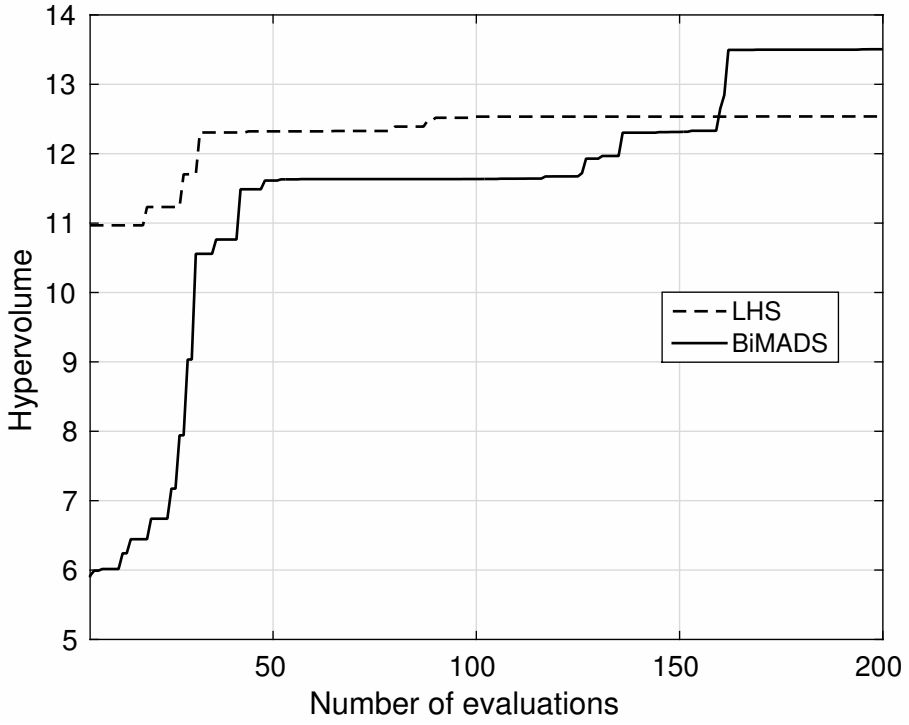


Fig. 5.8 Hypervolume profiles for FPGA implementations of the FGM: BiMADS vs LHS.

Table 5.2 Pareto-frontier approximation identified by BiMADS for FPGA implementations of FGM (corresponds to Figure 5.7). Designs are sorted according to their settling times in ascending order.

Design parameters					Design objectives	
T_s	N	N_{FGM}	q_{ratio}	N_{frac}	Settling time, s	Algorithm time, s
0.2132	9	230	0.20	25	8.3175	0.1321
0.2347	6	62	0.20	25	8.3329	0.1077
0.1870	4	40	0.20	17	8.7067	0.0596
0.3802	4	83	0.20	17	9.0659	0.0559
0.3802	4	62	0.20	17	9.1092	0.0558
0.5000	1	55	0.20	17	10.5620	0.0488
0.5000	1	34	0.20	16	10.6264	0.0467
0.4141	1	20	0.20	15	10.6878	0.0462
0.4141	1	20	0.20	13	12.4817	0.0396
0.5000	1	20	0.20	13	13.5297	0.0396
0.4785	1	20	0.20	13	13.8890	0.0396

5.5 Summary and future work

This chapter proposed automating predictive control design by employing systematic optimization. Several contradicting design objectives were discussed and a bi-objective optimization problem formulation was proposed. Decision variables for the resulting optimization problem include

both software and hardware parameters, which allows simultaneous optimization of both software and hardware and hence exploiting coupling between these components. It was shown that the bi-objective optimization-based design outperforms conventional exploration techniques and allows systematic investigation of resource-performance trade offs.

Future work includes exploiting the coupling between the nature of the design problem and design optimization algorithm. For instance, closed-loop cost function properties, which were recently studied in [11], can be used for building and updating a surrogate model and hence reducing the number of required simulations [55]. Another direction for further research is deriving analytical bounds for objective functions in order to eliminate unpromising designs without actual function evaluations.

Chapter 6

Conclusions and Future Work

This chapter will summarize the contributions and the impact of this thesis and outline possible future research directions.

6.1 Contributions

The key idea behind this thesis was to develop efficient hardware implementations of linear and nonlinear model predictive controllers. Both heuristic and systematic optimization-based design techniques were used in this work.

6.1.1 Automatic hardware code generation and testing for embedded optimization

An automatic FPGA code generation tool for embedded optimization using operator splitting methods was presented. The main features of the proposed code generator are:

- Smooth transition from a high-level description of the optimal control problem to deployment on FPGAs and processor-in-the-loop testing.
- Possibility of trading off resource usage again control performance by selecting design parameters at the code generation stage.

Processor-in-the-loop testing of the automatically generated code was performed using a new release of Protoip, which is another contribution of this thesis. A summary of the new release features:

- Possibility of prototyping heterogeneous implementations was added. The initial release of Protoip only dealt with pure FPGA implementations.
- Reliability of the communication channel between the PC and embedded platform for processor-in-the-loop simulation was improved by adding a new communication layer.
- An improved way of managing application projects was implemented to allow easier integration with code generation tools.

6.1.2 Nonlinear MPC on a heterogeneous computer

A framework for heterogeneous hardware implementation of a nonlinear MPC controller, based on an interior point algorithm, was proposed. The presented implementation provides flexibility of splitting the computational workload between a general-purpose CPU and a reconfigurable platform, i.e. FPGA. The main feature of the heterogeneous implementation is a method for scheduling sparse matrix-vector multiplication within an iterative linear system solvers to enable significant improvements in terms of computation time vs resource usage. According to the considered testcase an 18x memory saving compared to existing approaches and a 36x speedup over a pure software implementation can be achieved with the new implementation.

6.1.3 Automatic co-design of software and computational hardware for MPC

A systematic multi-objective optimization-based technique was applied to an MPC software and hardware co-design problem:

- A procedure was described for formulating an MPC design problem as a multi-objective optimization problem.
- Existing multi-objective optimization algorithms were reviewed with a focus on suitability for solving MPC software and hardware co-design problems.
- Two testcases were considered: CPU and FPGA-based implementations of predictive control using the fast gradient method.

6.2 Impact

The tools and methods proposed in this work can be used by engineers with various backgrounds both in academia and industry:

- "SPLIT + Protoip" toolchain bridges the gap between control/optimization engineers and computer architecture designers. The toolchain can be used for prototyping operator splitting algorithms on reconfigurable hardware without having a deep understanding of circuit design.
- The framework for heterogeneous hardware implementation of nonlinear MPC proposed in this work opens new possibilities of applying predictive control to fast dynamical systems in computational resource-limited environments.
- Systematic optimization-based MPC co-design approach presented in this thesis allows for faster design exploration compared to heuristic methods. This potentially reduces time to market when developing complex hardware and software solutions.

6.3 Future work

Some possible future research directions are outlined below.

6.3.1 Further automation of MPC design

In this work a general-purpose design optimization tool was applied to the MPC co-design problem. Further work will focus on exploiting the coupling between the nature of the design problem and design optimization algorithm. This can be achieved in several ways:

- The knowledge of MPC closed-loop cost function properties, e.g. continuity and monotonicity, which were recently studied in [11], can be incorporated into design optimization algorithms to speed up convergence.
- Tailored multi-fidelity optimization algorithms [31] can be developed for predictive controllers design. For many MPC design problems, objectives can be queried by performing simulations of different complexities, which allows trading off evaluation time and evaluation accuracy. For example, closed-loop performance can be evaluated for a limited set of initial conditions in order to a get rough estimate of performance; a wider range of initial conditions

must be considered in order to improve simulation accuracy sacrificing simulation time. Multi-fidelity optimization algorithms intend to exploit the trade off between accuracy and evaluation time in a systematic way.

6.3.2 Energy- and time-aware algorithms

Most of existing algorithms for real-time optimization, including the ones presented in Chapters 3 and 4 of this thesis, attempt to calculate fast and accurate solutions to open-loop optimal control problems without taking into consideration two important facts:

- The plant is evolving while computations are performed. Hence, a quick approximate solution might be better from a closed-loop perspective compared to accurate, but time-consuming solutions [48].
- Computational platforms consume energy with every iteration. For some applications, e.g. wireless sensor networks [1], the potential benefits of improving a satisfactory solution might be cancelled out by the relatively high energy consumption of a computational platform.

Efficient closed-loop oriented algorithms need to be developed for real-time optimization. These new algorithms shall take into account all components of the closed-loop cyber-physical systems, including computational hardware. The above challenges can be addressed by formulating the optimal control problem as a time-varying optimization problem [86] and deriving non-standard algorithm termination conditions that are likely to be based on directional derivatives of the objective function.

References

[1] Akyildiz, I., Su, W., Sankarasubramaniam, Y., and Cayirci, E. (2002). Wireless sensor networks: a survey. *Computer Networks*, 38(4):393 – 422.

[2] Amdahl, G. M. (2013). Computer architecture and Amdahl's law. *Computer*, 46(12):38–46.

[3] Andersson, J. (2013). *A General-Purpose Software Framework for Dynamic Optimization*. PhD thesis, Arenberg Doctoral School, KU Leuven, Department of Electrical Engineering (ESAT/SCD) and Optimization in Engineering Center, Kasteelpark Arenberg 10, 3001-Heverlee, Belgium.

[4] ARM (2011). *AMBA AXI and ACE Protocol Specification*.

[5] Atmel (2009). *8-bit AVR Microcontroller with 4/8/16/32K Bytes In-System Programmable Flash*. ATMEL, rev. 8161 edition.

[6] Audet, C. and Dennis, J. E. (2006). Mesh adaptive direct search algorithms for constrained optimization. *SIAM Journal on Optimization*, 17(1):188–217.

[7] Audet, C. and Dennis, J. E. (2009). A progressive barrier for derivative-free nonlinear programming. *SIAM Journal on Optimization*, 20(1):445–472.

[8] Audet, C., Savard, G., and Zghal, W. (2008). Multiobjective optimization through a series of single-objective formulations. *SIAM Journal on Optimization*, 19(1):188–210.

[9] Axehill, D. (2015). Controlling the level of sparsity in MPC. *Systems and Control Letters*, 76(0):1 – 7.

[10] Ayala, H., Sampaio, R., Muñoz, D. M., Llanos, C., Coelho, L., and Jacobi, R. (2016). Nonlinear model predictive control hardware implementation with custom-precision floating point operations. In *2016 24th Mediterranean Conference on Control and Automation (MED)*, pages 135–140.

[11] Bachtiar, V., Kerrigan, E. C., Moase, W. H., and Manzie, C. (2016). Continuity and monotonicity of the MPC value function with respect to sampling time and prediction horizon. *Automatica*, 63:330 – 337.

[12] Banjac, G., Goulart, P., Stellato, B., and Boyd, S. (2017). Infeasibility detection in the alternating direction method of multipliers for convex optimization. *Optimization Online*.

[13] Bellman, R. (1954). Dynamic programming and a new formalism in the calculus of variations. *Proceedings of the National Academy of Sciences of the United States of America*, 40(4):231–235.

[14] Bemporad, A., Morari, M., Dua, V., and Pistikopoulos, E. N. (2002). The explicit linear quadratic regulator for constrained systems. *Automatica*, 38(1):3 – 20.

[15] Betts, J. (2010). *Practical Methods for Optimal Control Using Nonlinear Programming*. Society for Industrial and Applied Mathematics, second edition.

[16] Boland, D. and Constantinides, G. A. (2008). An FPGA-based implementation of the MINRES algorithm. In *2008 International Conference on Field Programmable Logic and Applications*, pages 379–384. IEEE.

[17] Boland, D. and Constantinides, G. A. (2011). Optimizing memory bandwidth use and performance for matrix-vector multiplication in iterative methods. *ACM Trans. Reconfigurable Technol. Syst.*, 4(3):22:1–22:14.

[18] Boyd, S. P., Parikh, N., Chu, E., Peleato, B., and Eckstein, J. (2011). Distributed optimization and statistical learning via the alternating direction method of multipliers. *Foundations and Trends in Machine Learning*, 3(1):1–122.

[19] Buchstaller, D., Kerrigan, E., and Constantinides, G. (2012). Sampling and controlling faster than the computational delay. *IET Control Theory and Applications*, 6:1071–1079.

[20] Combettes, P. L., Condat, L., Pesquet, J. C., and Vu, B. C. (2014). A forward-backward view of some primal-dual optimization methods in image recovery. In *2014 IEEE International Conference on Image Processing (ICIP)*, pages 4141–4145.

[21] Conn, A., Scheinberg, K., and Vicente, L. (2009). *Introduction to Derivative-Free Optimization*. Society for Industrial and Applied Mathematics.

[22] Constantinides, G. A. (2009). Tutorial paper: Parallel architectures for model predictive control. *Proc. European control conference*, pages 138–143.

[23] Crockett, L., Elliot, R., and Enderwitz, M. (2014). *The Zynq Book: Embedded Processing with the Arm Cortex-A9 on the Xilinx Zynq-7000 All Programmable Soc*. Strathclyde Academic Media.

[24] Custodio, A., Emmerich, M., and Madeira, J. (2012). Recent developments in derivative-free multiobjective optimisation. *Computational Technology Reviews*, 5:1 – 30.

[25] Custódio, A. L., Madeira, J. F. A., Vaz, A. I. F., and Vicente, L. N. (2011). Direct multisearch for multiobjective optimization. *SIAM Journal on Optimization*, 21(3):1109–1140.

[26] Das, I. and Dennis, J. E. (1998). Normal-boundary intersection: A new method for generating the pareto surface in nonlinear multicriteria optimization problems. *SIAM Journal on Optimization*, 8(3):631–657.

[27] Davidon, W. C. (1991). Variable metric method for minimization. *SIAM Journal on Optimization*, 1(1):1–17.

[28] Debrouwere, F., Vukov, M., Quirynen, R., Diehl, M., and Swevers, J. (2014). Experimental validation of combined nonlinear optimal control and estimation of an overhead crane. *IFAC Proceedings Volumes*, 47(3):9617 – 9622. 19th IFAC World Congress.

[29] Ewe, C. T. (2005). Dual fixed-point: an efficient alternative to floating-point computation for DSP applications. In *International Conference on Field Programmable Logic and Applications*, pages 715–716.

[30] Fliess, M., Levine, J., and Rouchon, P. (1991). A simplified approach of crane control via a generalized state-space model. In *Proceedings of the 30th IEEE Conference on Decision and Control*, pages 736–741 vol.1.

[31] Forrester, A. I., Sóbester, A., and Keane, A. J. (2007). Multi-fidelity optimization via surrogate modelling. *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 463(2088):3251–3269.

[32] Garriga, J. L. and Soroush, M. (2010). Model predictive control tuning methods: A review. *Industrial & Engineering Chemistry Research*, 49(8):3505–3515.

[33] Gheribi, A. E., Harvey, J.-P., Bélisle, E., Robelin, C., Chartrand, P., Pelton, A. D., Bale, C. W., and Le Digabel, S. (2016). Use of a biobjective direct search algorithm in the process design of material science applications. *Optimization and Engineering*, 17(1):27–45.

[34] Giselsson, P. (2014). Improved fast dual gradient methods for embedded model predictive control. *IFAC Proceedings Volumes*, 47(3):2303 – 2309. 19th IFAC World Congress.

[35] Glowinski, R. and Le Tallec, P. (1989). *Augmented Lagrangian and Operator-Splitting Methods in Nonlinear Mechanics*. Society for Industrial and Applied Mathematics.

[36] Greenbaum, A. (1997). *Iterative Methods for Solving Linear Systems*. Society for Industrial and Applied Mathematics.

[37] Gulan, M., Takács, G., Nguyen, N. A., Olaru, S., Rodríguez-Ayerbe, P., and Rohal'-Ilkiv, B. (2017). Efficient embedded model predictive vibration control via convex lifting. *IEEE Transactions on Control Systems Technology*, PP(99):1–15.

[38] Hartley, E., Jerez, J., Suardi, A., Maciejowski, J. M., Kerrigan, E., and Constantinides, G. (2014). Predictive Control using an FPGA with Application to Aircraft Control. *IEEE Transactions on Control Systems Technology*, 22(3):1006–1017.

[39] Haselman, M., Beauchamp, M., Wood, A., Hauck, S., Underwood, K., and Hemmert, K. (2005). A comparison of floating point and logarithmic number systems for FPGAs. In *Field-Programmable Custom Computing Machines, 2005. FCCM 2005. 13th Annual IEEE Symposium on*, pages 181–190.

[40] Hindmarsh, A. C., Brown, P. N., Grant, K. E., Lee, S. L., Serban, R., Shumaker, D. E., and Woodward, C. S. (2005). SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers. *ACM Transactions on Mathematical Software (TOMS)*, 31(3):363–396.

[41] Janka, D., Kirches, C., Sager, S., and Wächter, A. (2016). An SR1/BFGS SQP algorithm for nonconvex nonlinear programs with block-diagonal hessian matrix. *Mathematical Programming Computation*, 8(4):435–459.

[42] Jerez, J. L., Constantinides, G. A., and Kerrigan, E. C. (2015). A low complexity scaling method for the Lanczos kernel in fixed-point arithmetic. *IEEE Transactions on Computers*, 64(2):303–315.

[43] Jerez, J. L., Kerrigan, E. C., and Constantinides, G. A. (2011). A condensed and sparse QP formulation for predictive control. In *2011 50th IEEE Conference on Decision and Control and European Control Conference*, pages 5217–5222.

[44] Jerez, J. L., Ling, K. V., Constantinides, G. A., and Kerrigan, E. C. (2012). Model predictive control for deeply pipelined field-programmable gate array implementation: algorithms and circuitry. *IET Control Theory & Applications*, 6:1029–1041(12).

[45] Jerez, J. L., Richter, S., Goulart, P. J., Constantinides, G. A., Kerrigan, E. C., and Morari, M. (2014). Embedded online optimization for model predictive control at megahertz rates. *Automatic Control, IEEE Transactions on*, 59(12):3238–3251.

[46] Kapinski, J., Deshmukh, J. V., Jin, X., Ito, H., and Butts, K. (2016). Simulation-based approaches for verification of embedded control systems: An overview of traditional and advanced modeling, testing, and verification techniques. *IEEE Control Systems*, 36(6):45–64.

[47] Kerrigan, E., Constantinides, G., Suardi, A., Picciau, A., and Khusainov, B. (2016). Computer architectures to close the loop in real-time optimization. In *Control and Decision Conference*, pages 4597–4611.

[48] Kerrigan, E. C. (2014). Co-design of hardware and algorithms for real-time optimization. In *European Control Conference*, pages 2484–2489.

[49] Khan, F. A., Hafeez, Z., Mirza, A., and u. Ain, Q. (2011). Design of FPGA based DAQ card using PCI express protocol. In *2011 IEEE 14th International Multitopic Conference*, pages 211–216.

[50] Khusainov, B., Kerrigan, E., and Constantinides, G. (2016). Multi-objective co-design for model predictive control with an FPGA. In *Control Conference (ECC), 2016 European*.

[51] Khusainov, B., Kerrigan, E. C., Suardi, A., and Constantinides, G. A. (2017). Nonlinear predictive control on a heterogeneous computing platform. In *IFAC World Congress 2017*.

[52] Kirk, D. (2012). *Optimal Control Theory: An Introduction*. Dover Books on Electrical Engineering. Dover Publications.

[53] Kleijnen, J. P. C. (2012). *Design and Analysis of Monte Carlo Experiments*, pages 529–547. Springer Berlin Heidelberg, Berlin, Heidelberg.

[54] Kurdi, Y. E., Gross, W. J., and Giannacopoulos, D. (2006). Hardware acceleration for finite element electromagnetics: Efficient sparse matrix floating-point computations with field programmable gate arrays. In *2006 12th Biennial IEEE Conference on Electromagnetic Field Computation*, pages 397–397.

[55] Le Digabel, S. (2011). Algorithm 909: NOMAD: Nonlinear optimization with the mads algorithm. *ACM Trans. Math. Softw.*, 37(4):44:1–44:15.

[56] Lofberg, J. (2004). YALMIP : a toolbox for modeling and optimization in MATLAB. In *2004 IEEE International Conference on Robotics and Automation (IEEE Cat. No.04CH37508)*, pages 284–289.

[57] Lopes, A. R. and Constantinides, G. A. (2008). *A High Throughput FPGA-Based Floating Point Conjugate Gradient Implementation*, pages 75–86. Springer Berlin Heidelberg, Berlin, Heidelberg.

[58] Lopes, A. R., Constantinides, G. A., and Kerrigan, E. C. (2008). A floating-point solver for band structured linear equations. In *2008 International Conference on Field-Programmable Technology*, pages 353–356.

[59] Lozano Santamaría, F. and Gómez, J. M. (2016). An algorithm for tuning NMPC controllers with application to chemical processes. *Industrial & Engineering Chemistry Research*, 55(34):9215–9228.

[60] Lucia, S., Navarro, D., Lucia, O., Zometa, P., and Findeisen, R. (2017). Optimized FPGA implementation of model predictive control for embedded systems using high level synthesis tool. *IEEE Transactions on Industrial Informatics*, PP(99):1–1.

[61] Mairal, J., Bach, F., and Ponce, J. (2014). Sparse modeling for image and vision processing. *Found. Trends. Comput. Graph. Vis.*, 8(2-3):85–283.

[62] Mathworks (2015). *MATLAB version 9.2 (R2017a)*. The Mathworks, Inc., Natick, Massachusetts.

[63] Messac, A., Ismail-Yahaya, A., and Mattson, C. (2003). The normalized normal constraint method for generating the Pareto frontier. *Structural and Multidisciplinary Optimization*, 25(2):86–98.

[64] Moore, G. E. (1998). Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1):82–85.

[65] Nesterov, Y. (2004). *Introductory lectures on convex optimization : a basic course*. Applied optimization. Kluwer Academic Publ., Boston, Dordrecht, London.

[66] Nocedal, J., Wächter, A., and Waltz, R. A. (2005). Adaptive barrier strategies for nonlinear interior methods. Technical Report RC 23563, IBM Watson Research Center, Yorktown Heights, NY, USA.

[67] Nocedal, J. and Wright, S. J. (2006). *Numerical Optimization*. Springer, New York, 2nd edition.

[68] Nurvitadhi, E., Mishra, A., and Marr, D. (2015). A sparse matrix vector multiply accelerator for support vector machine. In *2015 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, pages 109–116.

[69] Nvidia (2015). *Tesla K80 GPU Accelerator Board Specification*.

[70] O'Donoghue, B., Stathopoulos, G., and Boyd, S. (2013). A splitting method for optimal control. *IEEE Transactions on Control Systems Technology*, 21(6):2432–2442.

[71] Paige, C. C. and Saunders, M. A. (1975). Solution of sparse indefinite systems of linear equations. *SIAM Journal on Numerical Analysis*, 12(4):617–629.

[72] Parikh, N. and Boyd, S. (2010). Block splitting for large-scale distributed learning. In *Neural Information Processing Systems (NIPS) Workshop on Big Learning*.

[73] Patterson, D., Anderson, T., Cardwell, N., Fromm, R., Keeton, K., Kozyrakis, C., Thomas, R., and Yelick, K. (1997). A case for intelligent RAM. *IEEE Micro*, 17(2):34–44.

[74] Peyrl, H., Ferreau, H. J., and Kouzoupis, D. (2015). A hybrid hardware implementation for nonlinear model predictive control. *IFAC Conference on Nonlinear Model Predictive Control*, 48(23):87 – 93.

[75] Peyrl, H., Zanarini, A., Besselmann, T., Liu, J., and Boéchat, M.-A. (2014). Parallel implementations of the fast gradient method for high-speed MPC. *Control Engineering Practice*, 33(Supplement C):22 – 34.

[76] Piga, L. and Rigo, S. (2009). Comparing RTL and high-level synthesis methodologies in the design of a theora video decoder IP core. In *2009 5th Southern Conference on Programmable Logic (SPL)*, pages 135–140.

[77] Rawlings, J. and Mayne, D. (2009). *Model Predictive Control: Theory and Design*. Nob Hill Pub.

[78] Saad, Y. and Schultz, M. H. (1986). GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, 7(3):856–869.

[79] Sanchis, J., Martínez, M., Blasco, X., and Salcedo, J. V. (2008). A new perspective on multiobjective optimization by enhanced normalized normal constraint method. *Structural and Multidisciplinary Optimization*, 36(5):537–546.

[80] Scokaert, P. O. M., Mayne, D. Q., and Rawlings, J. B. (1999). Suboptimal model predictive control (feasibility implies stability). *IEEE Transactions on Automatic Control*, 44(3):648–654.

[81] Sedgewick, R. (1977). Permutation generation methods. *ACM Comput. Surv.*, 9(2):137–164.

[82] Shahriari, B., Swersky, K., Wang, Z., Adams, R. P., and de Freitas, N. (2016). Taking the human out of the loop: A review of Bayesian optimization. *Proceedings of the IEEE*, 104(1):148–175.

[83] Shahzad, A., Kerrigan, E. C., and Constantinides, G. A. (2012). A stable and efficient method for solving a convex quadratic program with application to optimal control. *SIAM Journal on Optimization*, 22(4):1369–1393.

[84] Shao, Y. S., Reagen, B., Wei, G. Y., and Brooks, D. (2014). Aladdin: A pre-RTL, power-performance accelerator simulator enabling large design space exploration of customized architectures. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 97–108.

[85] Shukla, H., Khusainov, B., Kerrigan, E., and Jones, C. (2017). Software and hardware code generation for predictive control using splitting methods. In *IFAC World Congress*.

[86] Simonetto, A., Mokhtari, A., Koppel, A., Leus, G., and Ribeiro, A. (2016). A class of prediction-correction methods for time-varying convex optimization. *IEEE Transactions on Signal Processing*, 64(17):4576–4591.

[87] Stathopoulos, G., Shukla, H., Szucs, A., Pu, Y., and Jones, C. N. (2016). Operator splitting methods in control. *Foundations and Trends® in Systems and Control*, 3(3):249–362.

[88] STMicroelectronics (2015). *STM32F303xB STM32F303xC product data*, 11 edition.

[89] Suardi, A., Constantinides, G. A., and Kerrigan, E. C. (2015). Software development kit for FPGA: A fast FPGA prototyping tool for embedded optimization. In *European Control Conference*.

[90] Takács, G., Batista, G., Gulan, M., and Rohal'-Ilkiv, B. (2016). Embedded explicit model predictive vibration control. *Mechatronics*, 36:54 – 62.

[91] Telen, D., Houska, B., Vallerio, M., Logist, F., and Impe, J. V. (2017). A study of integrated experiment design for nmpc applied to the droop model. *Chemical Engineering Science*, 160:370 – 383.

[92] Tseng, P. (1991). Applications of a splitting algorithm to decomposition in convex programming and variational inequalities. *SIAM Journal on Control and Optimization*, 29(1):119–138.

[93] Ullmann, F. (2011). FiOrdOs: A Matlab Toolbox for C-Code Generation for First Order Methods. Master's thesis, ETH Zurich.

[94] Vallerio, M., Van Impe, J., and Logist, F. (2014). Tuning of NMPC controllers via multi-objective optimisation. 61:38–50.

[95] Viana, F. A. (2013). Things you wanted to know about the latin hypercube design and were afraid to ask. In *10th World Congress on Structural and Multidisciplinary Optimization, Orlando, Florida, USA (cf. p. 69)*.

[96] Vouzis, P. D., Bleris, L. G., Arnold, M. G., and Kothare, M. V. (2009). A system-on-a-chip implementation for embedded real-time model predictive control. *IEEE Transactions on Control Systems Technology*, 17(5):1006–1017.

[97] Wächter, A. and Biegler, L. T. (2006). On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. *Mathematical Programming*, 106(1):25–57.

[98] Wright, S. (1997). *Primal-Dual Interior-Point Methods*. Society for Industrial and Applied Mathematics.

[99] Xilinx (2012). *Floating-Point Design with Vivado HLS*.

[100] Xilinx (2014a). *Vivado Design Suite User Guide. High-Level Synthesis*.

[101] Xilinx (2014b). *Vivado Design Suite. Vivado AXI Reference*.

[102] Xilinx (2014c). *Zynq-7000 All Programmable SoC First Generation Architecture*. v1.7.

[103] Xu, F., Chen, H., Gong, X., and Mei, Q. (2016). Fast nonlinear model predictive control on FPGA using particle swarm optimization. *IEEE Transactions on Industrial Electronics*, 63(1):310–321.

[104] Yamashita, A., Zanin, A., and Odloak, D. (2016a). Tuning of model predictive control with multi-objective optimization. *Brazilian Journal of Chemical Engineering*, 33(2):333–346.

[105] Yamashita, A. S., Zanin, A. C., and Odloak, D. (2016b). Tuning the model predictive control of a crude distillation unit. *ISA Transactions*, 60:178 – 190.

[106] Zavala, V. M. (2012). Real-time resolution of conflicting objectives in building energy management: An utopia-tracking approach.

[107] Zhao, J., Diehl, M., Longman, R., Bock, H., and Schlöder, J. (2004). Nonlinear model predictive control of robots using real-time optimization. In *Proceedings of the AIAA/AAS Astrodynamics Conference*, Providence, RI.

[108] Zhuo, L. and Prasanna, V. K. (2005). Sparse matrix-vector multiplication on FPGAs. In *Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field-programmable Gate Arrays*, FPGA '05, pages 63–74, New York, NY, USA. ACM.

Appendix A

Experimental setup data for Chapter 5

This appendix presents missing data for the experimental setup description in Sections 5.4.1.

Mass-spring-damper system parameters, starting from the mass attached to the wall:

Mass, kg	Spring constant, N/m	Damping constant, N · s/m
0.1966	0.5472	0.0509
0.2511	0.1386	0.1629
0.6160	0.1493	0.0487
0.4733	0.2575	0.1859
0.10517	0.08407	0.0700
0.4509	0.1966	0.5472
0.1629	0.0511	0.0386
0.1487	0.6160	0.1493
0.1859	0.4733	0.2575
0.0700	0.10517	0.08407

Initial conditions for closed-loop simulations, obtained with a random number generator with the range $[-0.2, 0.2]$:

Initial condition 1	Initial condition 2	Initial condition 3	Initial condition 4	Initial condition 5
0.0329	0.0163	0.1480	-0.0941	-0.0728
-0.1523	0.1759	0.0582	-0.0082	0.0557
0.0179	0.0589	0.0176	0.0884	0.0090
0.1975	-0.1125	-0.1577	-0.1561	-0.1746
-0.0382	-0.0207	-0.0537	0.1054	0.0512
0.1088	0.1731	0.1891	-0.1232	-0.1445
0.0785	-0.1625	0.0102	0.0121	0.1445
-0.0061	-0.0426	0.0686	0.0965	0.0080
-0.0609	-0.1400	0.0344	-0.0951	-0.1822
0.1020	-0.1029	-0.0230	0.0751	-0.0563
0.0945	-0.0421	0.0734	0.0816	-0.0231
-0.1922	-0.0677	-0.0303	-0.0919	-0.1212
0.1287	-0.0280	0.1551	-0.0435	0.1076
-0.0413	0.1234	0.1020	-0.0490	-0.1136
0.1162	0.1797	-0.0690	0.0685	-0.0245
0.1334	0.1075	-0.1331	0.1448	0.1959
0.0058	0.1537	0.0352	-0.1381	-0.1201
-0.0372	0.0995	0.1302	0.1160	-0.0726
0.0136	-0.1640	-0.1553	-0.1455	0.0715
-0.0019	-0.1241	-0.0020	-0.1410	-0.1780

Appendix B

Remaining details for Chapter 4

B.1 Remaining details for Algorithm 4

$$g(\theta_{[k]}) \coloneqq J\theta_{[k]} - d$$

$$W_{[k]} \coloneqq \Lambda_{[k]} G^{-1}(\theta_{[k]}),$$

where $\Lambda_{[k]}$ and $G(\theta_{[k]})$ are diagonal matrices containing the elements of $\lambda_{[k]}$ and $g(\theta_{[k]})$ respectively.

$$H_{[k]} \coloneqq \nabla_\theta f^T(\theta_{[k]}) \nabla_\theta f(\theta_{[k]}), \quad \mu = -\sigma \lambda_{[k]}^T g(\theta_{[k]})$$

$$r_{dual} \coloneqq \nabla_\theta g^T(\theta_{[k]}) G^{-1}(\theta_{[k]}) \mu e - \nabla_\theta f^T(\theta_{[k]}) \nabla_\theta f(\theta_{[k]}) - \nabla_\theta p^T(\theta_{[k]}) \nu, \quad r_{eq} \coloneqq -p(\theta_{[k]})$$

B.2 Synthesis results

Table B.1 Algorithm execution times with heterogeneous implementations. For all considered implementations the CPU clock rate is 667 MHz and the FPGA clock frequency is 166 MHz. $T_s = 100$ ms, $n_{iter} = 15$, t is the algorithm execution time.

N	P	SW	HG_1					HG_2					HG_3				
		t, ms	t, ms	LUT	FF	DSP	block RAM	t, ms	LUT	FF	DSP	block RAM	t, ms	LUT	FF	DSP	block RAM
1	1	72	34	2242	3242	8	11	20	6236	8429	20	26	9	8469	11723	43	33
2	1	201	94	2653	3726	10	12	49	6684	8793	20	26	19	8868	12071	43	33
2	2	-	94	2727	3859	10	15	49	6745	8972	23	33	16	8960	12136	43	40
3	1	406	180	2646	3836	10	11	92	6502	8758	20	26	31	8910	12084	43	33
3	3	-	180	3249	4485	15	19	91	7080	9202	26	40	24	9277	12264	43	47
4	1	654	288	2637	3868	10	11	140	6542	8817	20	26	47	8786	12143	43	33
4	4	-	290	3689	5059	20	25	141	7617	9852	31	47	32	9636	12472	43	54
5	1	967	421	2576	3817	10	12	202	6432	8762	20	27	65	8976	12062	43	34
5	5	-	421	4006	5714	25	31	201	8203	10706	36	55	42	10037	12851	43	62
6	1	1350	587	2547	3821	10	12	276	6439	8759	20	27	87	8671	12064	43	34
6	6	-	594	4424	6266	30	36	276	8705	1135	41	62	51	10542	13154	46	69
7	1	1812	781	2582	3849	10	13	362	6503	8814	20	28	112	9064	12172	43	35
7	7	-	783	4841	6931	35	41	360	9324	12100	46	69	62	11077	13914	51	76
8	1	2323	994	2564	3850	10	13	460	6499	8820	20	28	141	8820	12172	43	35
8	8	-	998	5200	7488	40	46	459	9820	12759	51	76	73	11489	14573	56	83
9	1	2877	1275	2590	3863	10	13	571	6503	8825	20	28	172	9093	12179	43	35
9	9	-	1275	5651	8045	45	51	571	10461	13401	56	83	84	12182	15222	61	90
10	1	3523	1524	2616	3865	10	13	694	6499	8828	20	28	205	8787	12174	43	35
10	10	-	1524	6005	8593	50	56	694	10906	14044	61	90	96	12693	15864	66	97

Appendix C

Optimal Scheduling for Sparse Matrix-Vector Multiplication on an FPGA

This appendix presents an optimal scheduling algorithm for pipelined sparse matrix-vector multiplication on a Field-Programmable Gate Array (FPGA) with a single Multiply-Accumulate (MAC) unit. The algorithm complexity scales linearly with the number of nonzero entries. A numerical example shows that the proposed algorithm outperforms a general-purpose integer optimization solver.

C.1 Introduction

Matrix-vector multiplication is a fundamental computational kernel that is widely used for various applications including optimization [51], signal processing [61] and machine learning [68]. Matrices arising from these applications often have a sparse structure, which is a result of loose coupling between underlying subsystems. Extensive studies have been conducted on accelerating sparse matrix-vector multiplication on Field-programmable Gate Arrays (FPGAs) in recent decades. Depending on the sparsity pattern regularity, hardware accelerators for matrix-vector multiplication can be classified into:

- *Regular sparsity pattern accelerators* that are tailored for a particular type of sparsity pattern. For example, [17] and [58] propose algorithms for banded matrix-vector multiplication.

- *Irregular sparsity pattern accelerators* that deal with general sparse matrices. This appendix is focused on this type of matrices.

The main bottleneck that prevents efficient acceleration for general sparse matrices is an irregular memory access pattern, which applies restrictions on efficient parallelization and pipelining. Based on the way how on-chip Random Access Memory (RAM) is utilized, general sparse matrix-vector multiplication accelerators can be classified into two classes:

- *External RAM-based* accelerators, e.g. [54] and [108], fetch data from external memory using FPGA local memory for temporary data storage. These implementations are often limited by I/O bandwidth.

- *On-chip FPGA RAM-based* accelerators store data locally on FPGAs, which becomes possible with modern FPGAs data storage capabilities. For instance, [57] presents a deeply pipelined implementation of matrix-vector multiplication, which, however, does not exploit sparsity. This work is focused on exploiting sparsity for sparse matrix-vector multiplication.

We present a scheduling algorithm for sparse matrix-vector multiplication on an FPGA, which allows dealing with sparse matrices irregularities in a systematic way. The proposed algorithm allows avoiding data dependencies so that the pipelining potential of floating point computational units is utilized efficiently.

C.2 Problem Statement

We consider the problem of sparse matrix-vector multiplication $y = Ax$, where $A \in \mathbb{R}^{k \times l}$ and the sparsity pattern of the matrix does not have any regularities. For example, the problem of interest

can be given by:

$$\underbrace{\begin{bmatrix} \bullet \\ \bullet \\ \bullet \\ \bullet \\ \bullet \\ \bullet \end{bmatrix}}_{y} = \underbrace{\begin{bmatrix} a_1^{[1]} & & a_1^{[2]} & a_1^{[3]} & a_1^{[4]} \\ a_2^{[1]} & & & & a_2^{[2]} \\ & a_3^{[1]} & & & \\ & & a_4^{[1]} & a_4^{[2]} & a_4^{[3]} \\ a_5^{[1]} & & & & \\ & & & a_6^{[1]} & \end{bmatrix}}_{A} \underbrace{\begin{bmatrix} \bullet \\ \bullet \\ \bullet \\ \bullet \\ \bullet \\ \bullet \end{bmatrix}}_{x}, \tag{C.1}$$

where $a_i^{[j]}$ is the j^{th} nonzero entry of the row i. The following problem-dependent parameters are introduced:

- N is the total number of nonzero entries in the matrix A.
- n is the number of nonzeros in the row(s) with the highest number of nonzeros.
- m is the number of rows that have the largest number of nonzeros.

For the problem (C.1): $N = 12$, $n = 4$ and $m = 1$. It is assumed that the matrix A is stored in the *Coordinate list (COO) format*, which, in contrast to the widely-used *Compressed Sparse Row (CSR) format*, allows accessing elements of a matrix in any required order. With the COO approach the matrices are stored in three vectors of equal size: row, column and value vectors. For the above example COO representation may be as follows:

$$\text{value} = \begin{bmatrix} a_1^{[1]} & a_1^{[2]} & a_1^{[3]} & a_1^{[4]} & a_2^{[1]} & a_2^{[2]} & a_3^{[1]} & a_4^{[1]} & a_4^{[2]} & a_4^{[3]} & a_5[1] & a_6^{[1]} \end{bmatrix} \tag{C.2}$$

$$\text{row} = \begin{bmatrix} 1 & 1 & 1 & 1 & 2 & 2 & 3 & 4 & 4 & 4 & 5 & 6 \end{bmatrix} \tag{C.3}$$

$$\text{column} = \begin{bmatrix} 1 & 3 & 4 & 5 & 1 & 5 & 2 & 3 & 4 & 5 & 1 & 4 \end{bmatrix} \tag{C.4}$$

Matrix-vector multiplication (C.1) can be implemented on an FPGA with a single Multiply-Accumulate (MAC) unit, which is composed of a floating point adder and multiplier (Figire C.1). It is assumed that input/output vectors and the matrix are stored in FPGA on-chip memory. Floating point units are often optimized for high throughput and fast clocking rates, while latency is considered as a secondary design objective. Hence efficient implementation of a certain algorithm

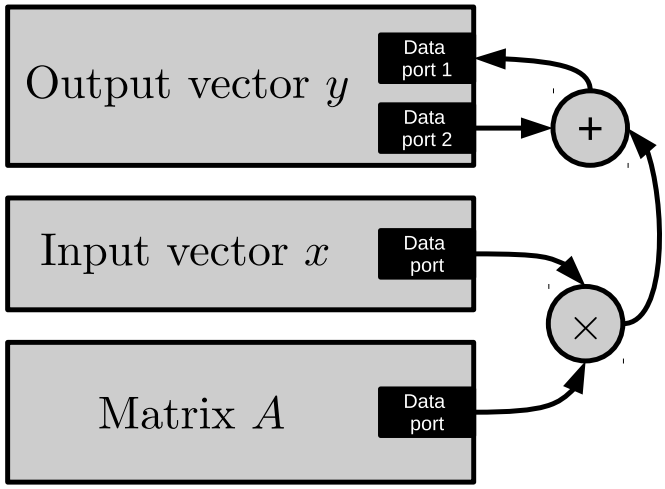


Fig. C.1 FPGA architecture for sparse matrix-vector multiplication with a single MAC unit.

requires exploiting pipelining possibilities, which is not always possible due to data dependencies. For the circuit of interest (Figure C.1), once a certain element from the output vector is read, it cannot be accessed again before the result of accumulation is written back to the corresponding address of the output vector memory. For instance if $a_1^{[1]}$ from the problem (C.1) is being processed, accumulation operation for the entries $a_1^{[2]}$, $a_1^{[3]}$ and $a_1^{[4]}$ cannot be started. Since accessing the same element of the output vector is associated with handling nonzero entries from the same row, operations have to be scheduled in such a way that the distance (in terms of clock cycles) between accessing the elements from the same row has to be maximized. This scheduling problem can be formalized as an integer optimization problem:

$$d^* := \max_{d,t(\cdot)} \quad d \tag{C.5a}$$

$$\text{subject to} \quad (d,t(\cdot)) \in X \tag{C.5b}$$

where d is a slack variable and $t(a_i^{[j]})$ is the number of the time instant at which processing of $a_i^{[j]}$ starts. The feasible set is given by

$$X := \begin{cases} (d, t(\cdot)) \quad \text{such that} & \\ |t(a_i^{[j]}) - t(a_k^{[l]})| \geq d & \forall a_i^{[j]} \in V, \forall a_k^{[l]} \in V : i = k, j \neq l \\ 1 \leq t(a_i^{[j]}) \leq N & \forall a_i^{[j]} \in V \\ t(a_i^{[j]}) \neq t(a_k^{[l]}) & \iff i \neq k, j \neq l \\ t(a_i^{[j]}) \in \mathbb{Z} & \forall a_i^{[j]} \in V \\ d \in \mathbb{Z} & \end{cases} \tag{C.6}$$

where V is the set of all nonzero elements. For the problem (C.1) $V = \{a_1^{[1]}, a_1^{[2]}, a_1^{[3]}, a_1^{[4]}, a_2^{[1]}, a_2^{[2]}, a_3^{[1]}, a_4^{[1]}, a_4^{[2]}, a_4^{[3]}, a_5^{[1]}, a_6^{[1]}\}$.

C.3 Scheduling Algorithm

This section will present an algorithm for solving optimization problem (C.5). Firstly, a relaxed optimization problem will be considered to derive an upper bound on objective d and, following this, a scheduling procedure for achieving the upper bound will be outlined.

C.3.1 A relaxed optimization problem for (C.5)

We make an assumption that there is only one row with the maximum number of nonzeros entries, i.e. $m = 1$. The following relaxed problem is considered:

$$\bar{d} := \max_{d, t(\cdot)} \quad d \tag{C.7a}$$

$$\text{subject to} \quad (d, t(\cdot)) \in \bar{X} \tag{C.7b}$$

The relaxed feasible set is given by:

$$\bar{X} := \begin{cases} (d, t(\cdot)) \quad \text{such that} & \\ |t(a_i^{[j]}) - t(a_k^{[l]})| \geq d & \forall a_i^{[j]} \in V, \forall a_k^{[l]} \in V : \boxed{i = k = r_{max}}, j \neq l \\ 1 \leq t(a_i^{[j]}) \leq N & \forall a_i^{[j]} \in V \\ t(a_i^{[j]}) \neq t(a_k^{[l]}) & \iff i \neq k, j \neq l \\ t(a_i^{[j]}) \in \mathbb{Z} & \forall a_i^{[j]} \in V \\ d \in \mathbb{Z} & \end{cases} \tag{C.8}$$

where r_{max} is the index of the row with the maximum number of nonzero entries.

While formulation (C.5) describes the problem of scheduling N operations into N time instants, relaxed formulation (C.7) corresponds to scheduling n operations into N time instants.

Lemma 1. *An upper bound on the value of the objective function of* (C.7) *is given by:*

$$d_{max} = \left\lfloor \frac{N-1}{n-1} \right\rfloor \tag{C.9}$$

Proof. The proof is based on contradiction. If d_{max} is not an upper bound for the objective of problem (C.7), then

$$(d_{max} + 1)(n-1) + 1 \not\leq N \tag{C.10}$$

$$\left(\left\lfloor \frac{N-1}{n-1} \right\rfloor + 1 \right)(n-1) + 1 \not\leq N \tag{C.11}$$

$$\left\lfloor \frac{N-1}{n-1} \right\rfloor + 1 \not\leq \frac{N-1}{n-1} \tag{C.12}$$

□

For the case $m > 1$ the upper bound would be given by:

$$d_{max} = \left\lfloor \frac{N-m}{n-1} \right\rfloor \tag{C.13}$$

The proof is also based based on contradiction and not presented here. The upper bound of the problem (C.7) can also serve as an upper bound for the problem (C.5):

$$d^* \leq \bar{d} \leq d_{max}, \tag{C.14}$$

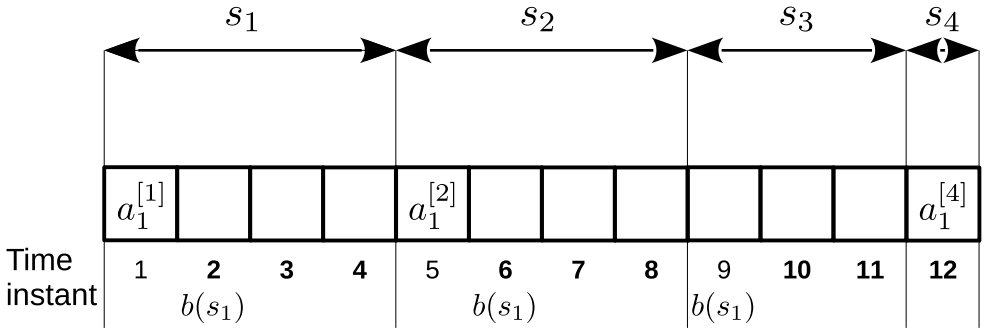


Fig. C.2 An example of an incomplete schedule for the problem (C.1).

which is a consequence of the fact that $X \subseteq \bar{X}$.

The next section will present an algorithm that allows achieving the above upper bound, i.e. (C.9) and (C.13).

C.3.2 Scheduling procedure

The proposed algorithm assumes that the schedule is divided into a sequence of n slots. The lengths of these slots are derived as follows:

- Slots 1 to $(N-m)\mathrm{mod}(n-1)$ have length $d_{max}+1$.
- Slots $((N-m)\mathrm{mod}(n-1))+1$ to $n-1$ have length d_{max}.
- Slot n has length m.

Time slots for the example (C.1), for which $d_{max}=3$, are presented on Figure C.2. The sequence of slots is denoted by S; s_i is the i^{th} slot of the sequence. Each member (i.e. slot) of the sequence S has the following properties:

- *Fullness.* A slot is considered to be *full* if an operation is scheduled for each time instant within that slot. Otherwise the slot is *nonfull*. Fullness property can be queried with a function $f : S_{set} \to \{0,1\}$, where S_{set} is the set of elements in the sequence S. The function f returns 1 if a set is full and 0 otherwise. For the example of an incomplete schedule in Figure C.2, the slot s_4 is full, while other slots are nonfull.
- *Beginning* of a slot is the earliest time instant for which operation is not scheduled. The beginning property is defined only for nonfull slots. The property can be queried with a function $b : S_{set} \to \mathbb{N}$.

Algorithm 6 Algorithm for scheduling sparse matrix-vector multiplication on an FPGA.

```
1: Require: sequences S and V_sort
2: flag = 1                                   ▷ exit flag for the outer while loop
3: j = 1                                      ▷ Counter over the sequence V_sort
4: while flag do
5:     for i = (1...n) do
6:         if not(f(s_i)) then                ▷ If the current slot is not full
7:             t(v_j) = b(s_i)                ▷ Schedule the current entry
8:             j = j + 1
9:             if j > N then                  ▷ If all elements are scheduled
10:                flag = 0                   ▷ exit the while loop
11:                break                      ▷ exit the for loop
12:            end if
13:        end if
14:    end for
15: end while
```

We introduce another sequence V_{sort}, which consists of the elements of V. The order of the elements is defined by the following rules:

- Elements from the same row are grouped together.
- Groups of elements from the same row are sorted according to their sizes in descending order.

v_i is the i^{th} element of the sequence V_{sort}. A possible sequence of matrix elements for the problem (C.1) is

$$V_{sort} = (a_1^{[1]}, a_1^{[2]}, a_1^{[3]}, a_1^{[4]}, a_4^{[1]}, a_4^{[2]}, a_4^{[3]}, a_2^{[1]}, a_2^{[2]}, a_3^{[1]}, a_5^{[1]}, a_6^{[1]}). \tag{C.15}$$

The scheduling procedure is presented in Algorithm 6. The main idea behind the algorithm is to iterate over the beginnings of time slots (s_i) by scheduling nonzeros elements in the order of sequence V_{sort}. Consider Figure C.3, which illustrates the algorithm's intermediate steps for the example (C.1). The algorithm is starts with an empty schedule, which is divided into time slots as shown in Figure C.2. Based on the way how these slots are constructed, the distance between consecutive slot beginnings is less than or equal to d_{max}. During the first iteration of the while loop four elements from V_{sort} are scheduled for the beginnings of the slots. Note that slot beginnings are shifted by one time instant after the iteration, which, does not affect the distance between beginnings. In the next iteration only three elements from V_{sort} are scheduled, since slot s_4 is already full. The third and fourth iteration are performed in a similar manner. With the above

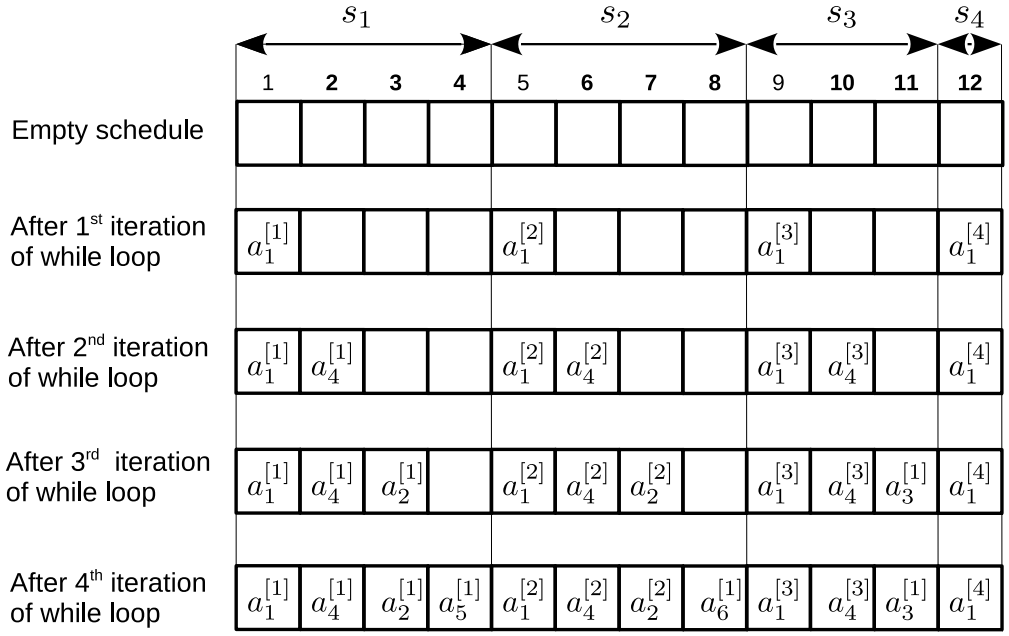


Fig. C.3 Algorithm 6 intermediate steps for the problem (C.1).

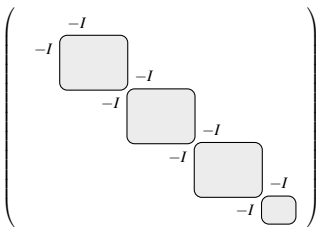


Fig. C.4 Sparsity pattern for the matrices that arise from optimal control, signal processing and differential equation-constrained optimization.

approach the distance between beginnings of the time slots is always maintained so that it never goes below d_{max}, which, taking into account (C.14), guarantees optimality.

C.4 Numerical example

Solving optimal control, signal processing and differential equation-constrained optimization problems often involve matrix-vector multiplications with the sparsity pattern presented in Figure C.4. Gray blocks correspond to separate trajectory points and negative identity matrices reflect the coupling between consecutive points, i.e. trajectory continuity. Significant memory savings can be achieved by considering the matrix as banded [17]. Further savings can be achieved by exploiting sparsity within gray blocks, which often share the same sparsity pattern. Handling negative identities can be narrowed down to data transfer operation with sign bit inversion. The remaining blocks

become decoupled in terms of accessing the elements of input and output vectors, hence, they can be computed in parallel.

We consider the matrix that arises from the problem of optimal control of the gantry crane [51] and has the structure presented in Figure C.4. Each of the 18×18 gray blocks of the matrix of interest has $N = 74$ nonzero elements; other problem parameters are $n = 5$ and $m = 6$. We attempt to solve the scheduling problem (C.5) in relation to the example from [51] using three different approaches that will be described below. All experiments were performed on the same desktop machine (Intel Xeon CPU E5-1650 v2 3.50GHz, 32GB of memory) allowing for a computational budget of 10 minutes.

- **Exhaustive search**, i.e enumerating all possible permutations. Heaps's algorithm [81] was able to generate ≈ 3 million permutations within the given budget of time, achieving $d = 4$.
- YALMIP's built-in **branch-and-bound** solver [56] achieved $d = 12$ in 36 seconds without further improvement during the remaining experiment time. This scheduling approach was used in [51].
- The **algorithm proposed in this appendix** solved the problem in 3 ms achieving the best possible distance of $d_{max} = 17$.

Although the proposed algorithm has a narrow scope of application, compared to exhaustive search and a branch-and-bound solver, the experiment shows that the considered problem of interest (C.5) can be solved more efficiently with the proposed approach, assuming the same computational resources. Figure C.5 demonstrates the impact of scheduling matrix-vector multiplication for the considered example. Pipelining without scheduling allows achieving a certain speedup compared to the unpipelined implementation. However, this speedup is limited by the adder latency. Systematic scheduling allows avoiding data dependencies and exploiting the throughput of the floating point adder, which results in significant improvements of the computation time.

C.5 Conclusions

This appendix presented an optimal scheduling algorithm for an efficiently pipelined implementation of sparse matrix-vector multiplication on low-cost FPGAs. The proposed approach assumes using a single pipelined MAC unit without introducing data-level parallelism. Further work could be

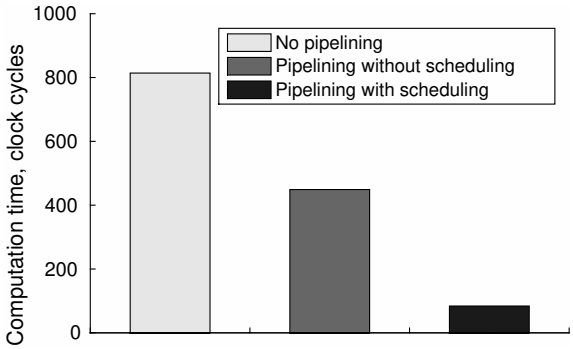


Fig. C.5 The impact of scheduling on computation time of a sparse matrix-vector multiplication (each gray block on Figure C.4) for the considered example. It is assumed that the latency for the addition and multiplication units are 6 and 5 clock cycles, respectively. The Figure is taken from [51].

focused on optimal partitioning of large sparse matrices so that the impact of partitioning on the possibility of pipelining is taken into account systematically.