A Survey of the Implementation of Linear Model Predictive Control on FPGAs ⋆

Ian McInerney * **George A. Constantinides** *
Eric C. Kerrigan *,**

* *Department of Electrical & Electronic Engineering, Imperial College London, SW7 2AZ London, UK, email:* {*i.mcinerney17,g.constantinides,e.kerrigan*}*@imperial.ac.uk*
** *Department of Aeronautics, Imperial College London, SW7 2AZ London, UK*

Abstract: Over the past 20 years, great strides have been made in the real-time implementation of linear MPC on FPGA devices. Starting from initial work, which demonstrated the benefits of embedding linear MPC onto FPGAs, recent work has shown sampling rates of more than 1 MHz are possible with FPGA-based implementations. This work surveys FPGA implementations of linear MPC, with a focus on the computational architecture. This includes the choice of number representation, the parallelizations exploited and the memory architecture. We discuss the transferability of those design choices to the FPGA implementation of nonlinear MPC, and provide some future research directions related to the implementation of MPC on FPGAs.

Keywords: linear MPC, embedded optimization, Field Programmable Gate Array (FPGA)

1. INTRODUCTION

Model predictive control (MPC) has grown increasingly popular over the past two decades, branching out from the chemical industry into other application areas. For instance, it has been applied to the control of spacecraft (Hartley and Maciejowski, 2013), aircraft (Hartley et al., 2014), and Atomic Force Microscopes (Jerez et al., 2013). These applications require very fast sampling rates, such as the Atomic Force Microscope, which needs control rates greater than 1 MHz for good closed loop performance.

Methods for solving MPC problems in real-time are grouped into two distinct sets: explicit MPC and implicit MPC. In explicit MPC, the state space is divided into sections, and the optimal control problem is solved for an initial condition in each section at design-time (Bemporad et al., 2002). At run-time, the process then consists of looking-up the appropriate feedback law in memory and applying the resulting input. While explicit MPC allows for very fast sampling rates, the memory usage can grow exponentially with the problem size, making it practical only for small problems.

In implicit MPC, the optimal control problem is solved at run-time. This consists of solving the optimization problem at each sampling instant, then applying the first computed control input to the system. This approach requires more run-time computation, since it utilizes an entire optimization solver instead of a lookup table search. Two main routes have been explored for implementing implicit linear MPC: CPU code generation and custom-designed hardware. CPU code generation is designed in an architecture-agnostic manner, and focuses on the generation of source code that is tailored to the optimization problem presented. The custom hardware architectures have been specifically designed with a focus on improving the performance of the algorithm.

To develop and implement these custom hardware architectures, designers utilize Field Programmable Gate Arrays (FPGAs). These devices contain an array of reconfigurable hardware cells, and an associated routing fabric to move signals between the cells (Constantinides, 2009). The advantage of FPGAs over CPUs stems from their ability to let designers create: custom number representations, specialized hardware for operators, and duplicate computational elements to parallelize computations.

A discussion of the computational architectures that can be applied in real-time control systems can be found in Kerrigan et al. (2015), but it focuses on a high-level description. This work describes the custom computational architectures for MPC that have been reported in the literature. Specifically, we focus on those which utilize FPGAs to implement implicit MPC with linear systems and a quadratic cost function.

Many FPGA implementations exist, with a summary of their characteristics provided in Table 1. We survey the architectural choices made in these implementations; for a survey of the optimization methods see Ferreau et al. (2017). We also discuss future research directions for both linear and nonlinear MPC on FPGA systems.

In Sections 2 and 3, the MPC formulations are presented and the three main algorithm classes are presented. In Section 4, number representation for MPC is discussed. In Section 5, the parallelizations at both an algorithmic and computational level are presented. In Section 6, memory

⋆ The support of the EPSRC Centre for Doctoral Training in High Performance Embedded and Distributed Systems (HiPEDS, Grant Reference EP/L016796/1) is gratefully acknowledged.

Table 1. FPGA implementation details for selected linear MPC solvers

Implementation Source	QP Form[1]	Algorithm[2]	Number Format[3]	Design Entry	Clock Frequency (MHz)	Matrix Storage[4]	QP Size[5]	Solver Time
Ling et al. (2006)	D	IP/CHOL	float32	Handel-C	20	BRAM	3/0/60	23.7 ms[6]
Ling et al. (2008)	D	IP/CHOL	float32	Handel-C	25	-	3/0/52	9.1 ms
Vouzis et al. (2009)	D	Newton	LNS	Verilog	50	BRAM	2/0/4	688 μs
Basterretxea and Benkrid (2011)	D	IP/CHOL	fixed	AccelDSP	20	BRAM	3/0/6	120 μs
Wills et al. (2011)	D	IP/CG	float16.6	VHDL	70	-	12/0/24	<200 μs
Yang et al. (2012)	D	ASM	float/fixed	Verilog	100	BRAM	3/0/6	20 μs
Wills et al. (2012)	D	ASM	float7	VHDL	70	BRAM	12/0/24	<30 μs
Peyrl et al. (2014)	D	FGM	fixed27.25	VHDL	120	-	15/0/30	0.49 μs
Jerez et al. (2014)	D	FGM	fixed	VHDL	400/230	BRAM	40/0/80	0.53/0.91 μs
Rubagotti et al. (2016)	D	DGP	fixed32.16	Simulink	100	DiRAM	20/0/208	239 μs
Liu et al. (2014)	S	IP/CHOL	float32	VHDL	200	BRAM	300/-/600	4 ms
Hartley et al. (2014)	S	IP/MINRES	float32	VHDL	250	BRAM	377/-/408	<12 ms
Jerez et al. (2014)	S	ADMM	fixed	VHDL	400/230	BRAM	216/-/172	4.9/8.52 μs
Dang et al. (2015)	S	ADMM	fixed32.17	-	-	-	120/80/250	215 ms[6]
Shukla et al. (2017)	S	AMA	float32	C	100	BRAM	384/-/-	900 μs[6]
Zhang et al. (2017)	S	ADMM	float32	VHDL	340	BRAM	204/-/300	30.1 μs

- Indicates data that was not reported

[1] D - Condensed (Dense) formulation, S - Uncondensed (Sparse) formulation

[2] ASM - Active-Set Method, IP - Interior-Point, FGM - Fast Gradient Method, ADMM - Alternating Direction Method of Multipliers, AMA - Alternating Minimization Algorithm, CHOL - Cholesky, CG - Conjugate Gradient, MINRES - Minimum Residual

[3] float32 - IEEE-754 single precision, LNS - Logarithmic Number System

[4] BRAM - Block RAM, DiRAM - Distributed RAM

[5] Decision Variables/Equality Constraints/Inequality Constraints

[6] Fully sequential implementation

architectures for MPC are presented. Finally, in Sections 7 and 8, trends in the implementation of linear MPC are presented, followed by a discussion of how these architectures can be extended in both linear and nonlinear MPC.

2. MPC PROBLEM FORMULATION

Implementations of linear MPC on FPGAs mostly use the constrained LQR formulation for Linear Time-Invariant (LTI) systems of the form

$$\min_{u,x} \|x_N\|_P^2 + \sum_{k=0}^{N-1} \left(\|x_k\|_Q^2 + \|u_k\|_R^2 \right) \tag{1a}$$

$$\text{s.t. } x_{k+1} = Ax_k + Bu_k,\ k = 0, \dots N-1 \tag{1b}$$

$$Fu_k \leq c_u,\ k = 0, \dots N-1 \tag{1c}$$

$$Dx_k \leq c_x,\ k = 1, \dots N \tag{1d}$$

for a horizon of length N where x_k and u_k are the states and inputs respectively at time sample k, A and B are the discrete-time state transition and input matrices respectively, and x_0 is a given estimate of the current state. The matrices F and D are the stage constraint matrices for the system states and inputs respectively, and the vectors c_u and c_x are the upper bounds for the stage constraints. The matrices Q, R, P are the weighting matrices for the system states, system inputs and final states respectively; when $Q, R, P \succeq 0$, the optimization problem (1) is convex.

The actual implementation of this optimization problem can be formulated in multiple ways. The sparse formulation contains both the state and the inputs as optimization variables, and leaves the system dynamics (1b) as equality constraints. In this formulation, all of the matrices in (1) are banded and sparse, and the number of non-zero elements in them grows linearly with the horizon length.

The condensed formulation removes the state variables x_k from the optimization problem by rewriting them as a function of only the inputs u_k. This allows the optimization problem to be transformed into a least-squares problem for the input variables, which moves the dynamics from the equality constraint (1b) to the Hessian matrix of the cost function (Maciejowski, 2002, §3). Though using the condensed form removes the equality constraints and reduces the number of optimization variables needed, this produces a dense Hessian matrix where the number of non-zero entries grows quadratically with the horizon length.

3. QUADRATIC PROGRAMMING ALGORITHMS

To solve (1), FPGA implementations have utilized one of three main convex optimization algorithm classes: Interior-Point Methods, Active-Set Methods, or First-Order Methods. Each algorithm has several variants, and many have been implemented on FPGAs, as illustrated in Figure 1.

Interior-Point Methods (IPMs) were some of the original methods applied to solve (1) by Ling et al. (2006) and subsequent authors. In IPM, the optimal solution of (1) is found by solving the nonlinear KKT system of equations. This system is solved using an iterative Newton's method, with each iteration consisting of three main steps:

(1) Update/linearize the KKT system.
(2) Solve the linear KKT system for a step direction.
(3) Update the current guess with the step direction.

The main computational burden is solving the linear system in Step 2. Work by Rao et al. (1998) has shown that exploiting the structure of the matrices arising in the LTI problem leads to faster and more scalable solvers.

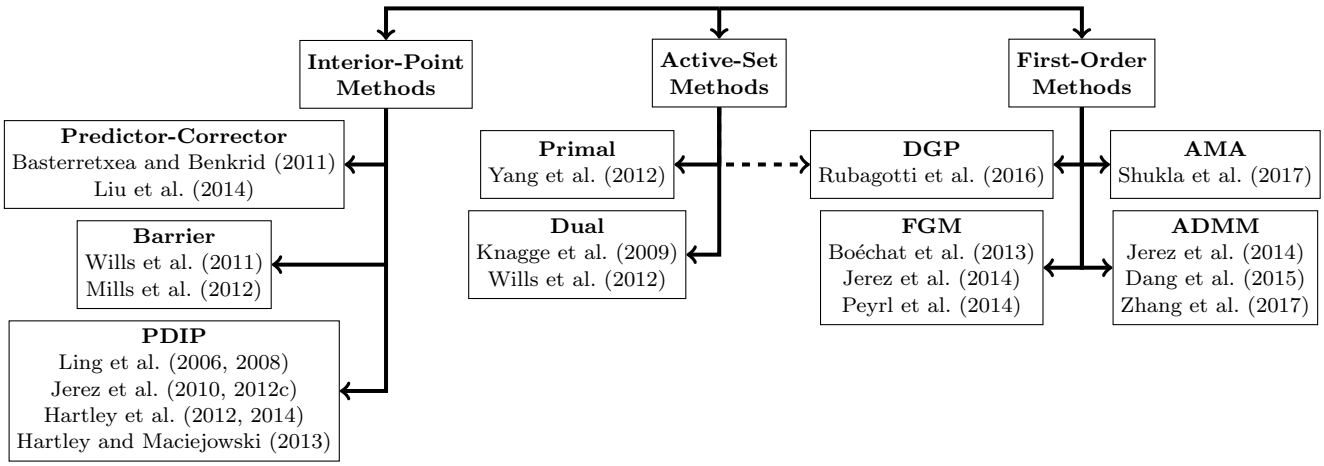

Fig. 1. Taxonomy of the various optimization algorithm implementations discussed in this paper.

The second type of algorithm that has been implemented on FPGA systems is the Active-Set Method (ASM). In ASM, the algorithm finds the optimal point by solving a sequence of equality-constrained subproblems to locate the set of inequality constraints that are active at optimality. In each iteration, inequality constraints are converted to equality constraints based upon whether they have been violated in the previous iteration.

A comparison between IPM and ASM on FPGAs was carried out by by Lau et al. (2009). The results show that for small problem sizes, ASM will take fewer iterations compared to the fixed cost of the IPM. However, the number of interior-point iterations required stays constant as the problem size grows, while the iterations required for ASM tends to grow approximately linearly with the number of optimization variables and constraints. They determined that for larger problems, IPM is better to use.

More recently, FPGA implementations have used first-order methods such as Nesterov's Fast Gradient Method (FGM), or the Alternating Direction Method of Multipliers (ADMM). These methods utilize only first-order information (e.g. the gradient) in their computations, and generally have three main steps:

(1) Compute a search direction.
(2) Compute the step size and apply the search direction.
(3) Project onto the feasible set.

The main linear algebra operation in these methods is a matrix-vector multiplication in Step 1 (as opposed to the linear system solver in IPM or ASM). However, the projection operation in Step 3 can become computationally complex depending on the feasible set. If the constraint sets (1c), (1d) are purely upper and lower bounds, then the projection operation is a simple variable saturation. However, if $F \neq I$ or $D \neq I$ then the projection operation may require the solution of its own QP.

4. NUMBER REPRESENTATION

An important feature of FPGAs compared to CPUs is their support for non-standard data types in the calculations. This allows the designer to create hardware tailored to different word lengths in fixed-point, and different mantissa and exponent sizes in floating-point. These custom data types reduce the amount of device area consumed by the individual computational resources, allowing for more parallelism (Constantinides, 2009). The choice of data type introduces two types of errors: computational errors and representation errors.

4.1 Computational Errors

Computational errors manifest themselves in three ways:

- *Overflow:* The magnitude of the requested number is larger than the maximum representable number.
- *Underflow:* The magnitude of the requested number is smaller than the smallest representable number.
- *Round-off:* Errors that occur because the number requires more fractional places than available.

The optimization algorithms discussed in Section 3 were developed using either infinite precision or the standard data types (e.g. IEEE-754 floating-point). Changing the algorithms to another data type will introduce different errors in the computation. For instance, changing from floating-point to fixed-point introduces different round-off errors in the computations. To guarantee the algorithm works with the new data types, new proofs of convergence and stability are needed, such as those for the fast gradient method in Jerez et al. (2013) and ADMM in Jerez et al. (2014).

An additional concern for fixed-point implementations is overflow errors. In general, bounding values in algorithms is difficult due to the nonlinear nature of the computations. However, bounds do exist for some methods, such as the fast gradient method (Jerez et al., 2013), ADMM (Dang et al., 2015) and dual gradient projection (Patrinos et al., 2013). These bounds then lead to rules for the sizing of the integer and fractional components in fixed-point to guarantee no overflow errors occur inside the algorithms.

Other algorithms, such as IPM, do not have bounds for every variable in the computation. Instead, bounds on variables in parts of the algorithm have been created. For instance, when the Minimum Residual (MINRES) solver is used for Step 2 of the IPM, Jerez et al. (2012a) present a scaling method that guarantees all signals in the MINRES solver will stay below precomputed bounds, allowing for a fixed-point implementation of MINRES.

4.2 Representation Errors

In addition to examining the effect of the data type on the algorithm's convergence, designers must be conscious of the effect this has on the optimization problem itself. For instance, the choice of the representation could cause the optimization problem to lose convexity or feasibility with respect to the original problem.

For MPC specifically, experiments by Longo et al. (2014) show that the discretization is affected by the data type. Their experiments show that the usual shift-form discretization method performs very poorly when used inside an interior-point method with small data types (e.g. 5-bit floating-point). Instead, they propose using the delta-domain discretization method. This then provides closed-loop control using 5-bit floating-point that is equivalent to the control using double precision (52-bit) floating-point.

5. PARALLELIZATION OPPORTUNITIES

Opportunities for parallelization occur at two distinct levels: algorithmic and computational. Algorithmic parallelizations are those inherent in the structure of the algorithm, while computational level parallizations exist at the level of the arithmetic computations.

5.1 Algorithmic Level

In this work, algorithmic level parallelization refers to the ability to parallelize the overall algorithm steps inside a single iteration. The existence of these parallelizations is highly dependent upon the structure of the problem, and of the algorithm itself.

In IPM implementations on FPGAs (such as Jerez et al. (2012b); Liu et al. (2014); Hartley et al. (2014)), the main computational bottleneck is located inside step 2 (the linear system solver), so attempting to parallelize the computations in Steps 1 and 3 provided no noticeable performance increase. Therefore, these implementations left Steps 1 and 3 with minimal parallelization and focused the development effort on the computational level parallelization inside Step 2.

More recent FPGA implementations have examined the parallelizations possible in first-order methods, such as FGM or ADMM. Work by Jerez et al. (2014) demonstrates that, given an appropriate projection operation, there is no data-dependence between the optimization variables in an iteration. This allows for the processing of each variable in parallel by duplicating the computational component containing a Marix-Vector Multiplier (MVM), projection block, and associated glue logic.

This parallelization is highly dependent upon the projection operator though, and the reported architectures utilize box constraints on the variables to allow for this exploitation. The introduction of constraints such as $Cx \leq d$ will introduce data dependencies in the projection, and make parallelization highly dependent upon the structure of the matrix C. Adding soft constraints to the MPC problem was done in Jerez et al. (2014), but to maximize the parallelization in the ADMM implementation there had to be one slack variable per soft constraint.

5.2 Computational Level

Many of the parallelizations that have been exploited in the FPGA implementations have been at the computational level. This level consists of the low-level linear algebra routines that compose the algorithm, such as the linear system solvers, matrix factorizations and matrix-vector multiplies.

An initial survey of parallelizing the linear system solvers was done by Lopes et al. (2009), where they explored the implementation of the Conjugate Gradient method using deeply pipelined inner-product units. Subsequent implementations by Jerez et al. (2012b) and Hartley et al. (2014) explored the implementation of the MINRES solver to speed-up Step 2 inside of the interior-point method by parallelizing and pipelining the operations.

Parallelization of the MVM was used in nearly every implementation, but there were two main methods used: column-sweep and row-sweep. Column-sweep parallelism was implemented in Rubagotti et al. (2016), and consisted of a multiply-accumulate (MAC) module for each row of the matrix. This processes a column at a time, and will not produce a complete result until n clock cycles for an $m \times n$ matrix.

Row-sweep parallelism was implemented in Wills et al. (2011) and Jerez et al. (2014), and consists of multiple units that compute the dot products between rows of the matrix and the vector. There is one multiplier per vector element, followed by an adder tree to combine all the results. This adder tree can have registers inserted between levels to reduce the critical path and allow for pipelining of different operations. A fully pipelined implementation (registers between every level) of a single MVM unit can produce the complete result in $m + \lceil \log_2 n \rceil$ clock cycles.

6. MEMORY ORGANIZATION

The architecture of the memory system consists of two main components: locality and the data storage pattern. Locality refers to how close the memory cells are to the computational elements, while the data storage pattern is how the data/matrices are arranged inside the memory cells.

6.1 Locality

There are three major memory regions that are available for FPGA implementations: Distributed RAM (DiRAM), Block RAM (BRAM), and off-chip memory. In the MPC implementations examined, none of them utilized off-chip memory, and all but one utilized the BRAM to store the main coefficient matrix (the Hessian), and the DiRAM for various computational products.

The remaining implementation (Rubagotti et al., 2016) stores the Hessian matrix in DiRAM to have high locality with the MVM elements. For the active-set implementation done by Wills et al. (2012), they experimented with placing the Hessian matrix in the DiRAM and the BRAM. They reported results that show switching from BRAM to DiRAM will almost double the resource usage of the FPGA.

6.2 Storage Pattern

The storage pattern that an implementation utilizes is driven by both the choice of the MPC problem formulation and the computational architecture. When designing the storage pattern in relation to the computational architecture, the primary objective is to present the data to the computational units with the smallest latency. For instance, for the column-sweep MVM implementation in Rubagotti et al. (2016), the Hessian matrix is broken into rows for storage at each MAC (each row in separate memory areas), allowing for each MAC to be fed with a new coefficient simultaneously. Alternatively, the work of Jerez et al. (2012b) places each column of the Hessian in different BRAM units to efficiently feed the coefficients to the column-sweep MVM unit.

The chosen MPC problem formulation plays a key role in the scaling of the storage pattern as problem sizes change. For instance, in the FPGA implementations using the condensed formulation with a full Hessian matrix, the memory usage of the Hessian scales quadratically with the horizon length.

Alternatively, choosing the sparse formulation allows for the usage of the Compressed Diagonal Storage (CDS) method to represent the Hessian matrix. As detailed in Boland and Constantinides (2010), CDS exploits the banded structure of an $m \times n$ matrix to only store the diagonals that contain values (those inside the bandwidth r). This is accomplished by turning each diagonal of the original matrix into a column of the CDS matrix, which then has dimensions $m \times r$. The CDS matrix then grows linearly with the horizon length. The CDS storage method was utilized in Jerez et al. (2012b), where it decreased memory usage by at least an order of magnitude.

Further structure in the sparse LTI MPC problem can be exploited to realize gains of over 75% compared to the normal CDS implementation. Specifically, Jerez et al. (2012b) shows that the CDS matrix for the LTI case contains many identical rows (with mostly zeros and ones). By storing only one copy of that row and then mapping the appropriate memory addresses to it when needed, memory use can be reduced. Additional savings can be found by exploiting the block structure of the matrix, so only one copy of a block is stored and then accessed through an appropriately designed memory system.

7. OVERALL TRENDS

The implementation of linear MPC on FPGAs has seen substantial improvement over the past 15 years, with a diverse set of implementations that have pushed the computational time to sub-microsecond ranges. By examining the implementation details of many different solvers (provided in Table 1), several patterns emerge.

The first is that the recent speed-up in solver times has been accompanied by a switch to first-order methods, such as FGM for a condensed problem and ADMM for the sparse problem. This switch shows an order of magnitude difference in the number of clock cycles required for the computation, as can be seen in the comparison presented in Figure 2.

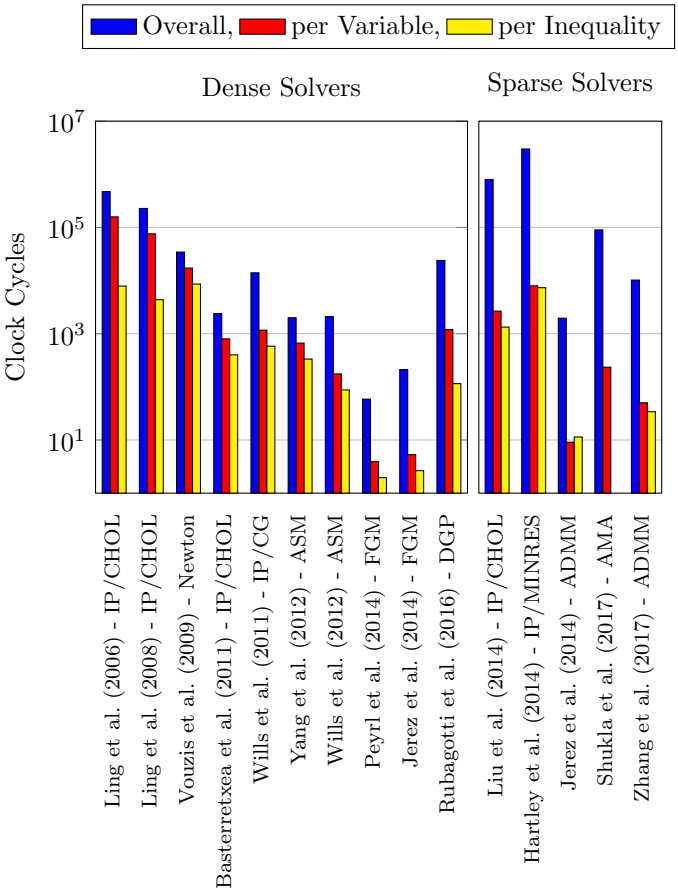

Fig. 2. Cycles required for the computations. Per variable (inequality) values are the total clock cycles divided by the number of variables (inequality constraints).

Another pattern that emerges is that, unlike many initial implementations that were based on a high-level language, such as Simulink or Handel-C, recent implementations have been coded directly in a low-level Register-Transfer Level (RTL) language, such as VHDL or Verilog.

8. FUTURE DIRECTIONS

8.1 Continuing with Linear MPC

As identified earlier, many of the recent results that have produced large reductions in solver time have utilized RTL languages. Implementations at this level can be difficult for control designers to work with, and may not be widely accepted. To encourage acceptance by the control designer, a simplified design-flow is needed, similar to that provided by the many code generation utilities for CPUs.

Initial work done by Shukla et al. (2017) has led to the development of the tool SPLIT, which uses MATLAB to generate C code tailored to the Vivado High Level Synthesis (HLS) toolset. Other work conducted by Lucia et al. (2018) utilized existing CPU code generation tools with FPGA HLS tools to explore the software-hardware co-design space. The reported results indicate that the software-hardware co-design space is very rich, and that care should be taken when choosing design parameters (e.g. level of parallelization).

Automatically creating MPC implementations that achieve optimality in this space is a new research area that has attracted attention. Work by Khusainov et al. (2016, 2017a) has started to explore the automated co-design of MPC implementations, but a lot of work remains in the identification of the design criterion, and also the optimization algorithms to employ.

8.2 Moving to Nonlinear MPC

Implementing nonlinear MPC in real-time systems is another young research field that has risen to prominence in recent years. A survey of the algorithms that can be used and their associated details can be found in Gros et al. (2016). Work in Peyrl et al. (2015) and Khusainov et al. (2017b) has started to explore the implementation of these algorithms on an FPGA system. While several pieces of the linear MPC implementation carry over, there are many unanswered questions remaining.

When implementing a nonlinear MPC tracking/regulator problem, much of the computational level parallelism (e.g. MVM architectures) can still be utilized, and even some of the algorithmic parallelism carries over (e.g. separability across variables in FGM). However, there are new parts in the algorithms (e.g. condensing and linearization) where the parallelization opportunities and scalability have not yet been explored and understood in an FPGA framework.

Additionally, the behavior of nonlinear MPC algorithms in arbitrary-precision number representations is important. As shown in the linear case, the use of the standard shift-form discretization can produce poor performance with small arbitrary-precision number representations. There are many other discretization methods that can be used with nonlinear MPC though, so the performance of the various methods under arbitrary-precision should be studied and understood to reveal any possible benefits or caveats.

Another important question centers on memory structure and usage. While the nonlinear sparse formulation can still utilize the CDS structure, the amount of constant/similar data in the matrix decreases drastically. Additionally, current trends in nonlinear implementation suggest linearizing on a CPU then passing data to the FPGA. This introduces new questions about how to best structure the memory and its transactions to have both computational efficiency in the control algorithm and efficient transactions in the matrix update stage.

9. CONCLUSIONS

Overall, a diverse set of literature exists on the implementation of linear MPC on FPGAs. Recent implementations have shown that with the appropriate parallelizations at the computational and algorithmic level, solvers completing in less than 1μs are possible using first-order methods.

The knowledge gained from the linear implementations can be carried forth into the nonlinear realm, but it is not sufficient. New research needs to be conducted into how other parts of the algorithms scale/parallelize, how number representation affects the algorithm and how memory architectures should be designed.

REFERENCES

Basterretxea, K. and Benkrid, K. (2011). Embedded high-speed Model Predictive Controller on a FPGA. In *Proceedings of the 2011 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, 327–335. San Diego, CA.

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

Boéchat, M.A., Liu, J., Peyrl, H., Zanarini, A., and Besselmann, T. (2013). An Architecture for Solving Quadratic Programs with the Fast Gradient Method on a Field Programmable Gate Array. In *2013 21st Mediterranean Conference on Control and Automation (MED)*, 1557–1562. Platanias-Chania, Crete, Greece.

Boland, D. and Constantinides, G.A. (2010). Optimising Memory Bandwidth Use for Matrix-Vector Multiplication in Iterative Methods. In *Proceedings of the International Symposium on Applied Reconfigurable Computing*, 169–181. Bangkok, Thailand.

Constantinides, G.A. (2009). Tutorial Paper: Parallel Architectures for Model Predictive Control. In *Proceedings of the 2009 European Control Conference (ECC)*, 138–143. Budapest, Hungary.

Dang, T.V., Ling, K.V., and Maciejowski, J.M. (2015). Embedded ADMM-based QP solver for MPC with polytopic constraints. In *2015 European Control Conference (ECC)*, 3446–3451. Linz, Austria.

Ferreau, H.J., Almér, S., Verschueren, R., Diehl, M., Frick, D., Domahidi, A., Jerez, J.L., Stathopoulos, G., and Jones, C. (2017). Embedded Optimization Methods for Industrial Automatic Control. In *Proceedings of the 20th IFAC World Congress*, 13194–13209. Toulouse, France.

Gros, S., Zanon, M., Quirynen, R., Bemporad, A., and Diehl, M. (2016). From linear to nonlinear MPC: bridging the gap via the real-time iteration. *International Journal of Control*, 1–19.

Hartley, E.N., Jerez, J.L., Suardi, A., Maciejowski, J.M., Kerrigan, E.C., and Constantinides, G.A. (2014). Predictive Control Using an FPGA With Application to Aircraft Control. *IEEE Transactions on Control Systems Technology*, 22(3), 1006–1017.

Hartley, E.N. and Maciejowski, J.M. (2013). Predictive control for spacecraft rendezvous in an elliptical orbit using an FPGA. In *2013 European Control Conference (ECC)*, 1359–1364. Zurich.

Jerez, J.L., Constantinides, G.A., and Kerrigan, E.C. (2012a). Towards a fixed point QP solver for predictive control. In *51st IEEE Conference on Decision and Control (CDC)*, 675–680. Maui, HI.

Jerez, J.L., Goulart, P.J., Richter, S., Constantinides, G.A., Kerrigan, E.C., and Morari, M. (2013). Embedded Predictive Control on an FPGA using the Fast Gradient Method. In *2013 European Control Conference (ECC)*, 3614–3620. Zurich.

Jerez, J.L., Goulart, P.J., Richter, S., Constantinides, G.A., Kerrigan, E.C., and Morari, M. (2014). Embedded Online Optimization for Model Predictive Control at Megahertz Rates. *IEEE Transactions on Automatic Control*, 59(12), 3238–3251.

Jerez, J.L., Ling, K.V., Constantinides, G.A., and Kerrigan, E.C. (2012b). Model predictive control for deeply

pipelined field-programmable gate array implementation: algorithms and circuitry. *IET Control Theory and Applications*, 6(8), 1029 – 1041.

Kerrigan, E.C., Constantinides, G.A., Suardi, A., Picciau, A., and Khusainov, B. (2015). Computer Architectures to Close the Loop in Real-time Optimization. In *54th IEEE Conference on Decision and Control (CDC)*, 4597–4611. Osaka, Japan.

Khusainov, B., Kerrigan, E.C., and Constantinides, G.A. (2016). Multi-objective Co-design for model predictive control with an FPGA. In *2016 European Control Conference (ECC)*, 110–115. IEEE, Aalborg, Denmark.

Khusainov, B., Kerrigan, E.C., and Constantinides, G.A. (2017a). Automatic Software and Computing Hardware Co-design for Predictive Control. *arXiv preprint*, 1–10.

Khusainov, B., Kerrigan, E.C., Suardi, A., and Constantinides, G.A. (2017b). Nonlinear predictive control on a heterogeneous computing platform. In *Proceedings of the 20th IFAC World Congress*. Toulouse, France.

Knagge, G., Wills, A., Mills, A., and Ninness, B. (2009). ASIC and FPGA implementation strategies for Model Predictive Control. In *Proceedings of the 2009 European Control Conference (ECC)*, 144–149. Budapest, Hungary.

Lau, M.S.K., Yue, S.P., Ling, K.V., and Maciejowski, J.M. (2009). A Comparison of Interior Point and Active Set Methods for FPGA Implementation of Model Predictive Control. In *Proceedings of the 2009 European Control Conference (ECC)*, 3–8. Budapest, Hungary.

Ling, K.V., Yue, S.P., and Maciejowski, J.M. (2006). A FPGA Implementation of Model Predictive Control. In *2006 American Control Conference (ACC)*, 1930–1935. Minneapolis, MN, USA.

Ling, K.V., Wu, B.F., and Maciejowski, J. (2008). Embedded Model Predictive Control (MPC) using a FPGA. In *Proceedings of the 17th IFAC World Congress*, 15250–15255. Seoul, Korea.

Liu, J., Peyrl, H., Burg, A., and Constantinides, G.A. (2014). FPGA implementation of an interior point method for high-speed model predictive control. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, 1–8. Montreal, QC, Canada.

Longo, S., Kerrigan, E.C., and Constantinides, G.A. (2014). Constrained LQR for low-precision data representation. *Automatica*, 50(1), 162–168.

Lopes, A.R., Shahzad, A., Constantinides, G.A., and Kerrigan, E.C. (2009). More Flops or More Precision? Accuracy Parameterizable Linear Equation Solvers for Model Predictive Control. In *Proceedings of the 17th IEEE Symposium on Field Programmable Custom Computing Machines (FCCM '09)*, 209–216. Napa, CA.

Lucia, S., Navarro, D., Lucia, O., Zometa, P., and Findeisen, R. (2018). Optimized FPGA Implementation of Model Predictive Control for Embedded Systems Using High Level Synthesis Tool. *IEEE Transactions on Industrial Informatics*, 14(1), 137–145.

Maciejowski, J.M. (2002). *Predictive Control with Constraints*. Pearson Education Limited, Essex, UK.

Mills, A., Wills, A.G., Weller, S.R., and Ninness, B. (2012). Implementation of linear model predictive control using a field-programmable gate array. *IET Control Theory and Applications*, 6(8), 1042–1054.

Patrinos, P., Guiggiani, A., and Bemporad, A. (2013). Fixed-point dual gradient projection for embedded model predictive control. In *2013 European Control Conference (ECC)*. Zurich, Switzerland.

Peyrl, H., Ferreau, H.J., and Kouzoupis, D. (2015). A Hybrid Hardware Implementation for Nonlinear Model Predictive Control. In *5th IFAC Conference on Nonlinear Model Predictive Control*, 87–93. Seville, Spain.

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, 22–34.

Rao, C.V., Wright, S.J., and Rawlings, J.B. (1998). Application of Interior-Point Methods to Model Predictive Control. *Journal of Optimization Theory and Applications*, 99(3), 723–757.

Rubagotti, M., Patrinos, P., Guiggiani, A., and Bemporad, A. (2016). Real-time model predictive control based on dual gradient projection: Theory and fixed-point FPGA implementation. *International Journal of Robust and Nonlinear Control*.

Shukla, H.A., Khusainov, B., Kerrigan, E.C., and Jones, C.N. (2017). Software and Hardware Code Generation for Predictive Control Using Splitting Methods. In *Proceedings of the 20th IFAC World Congress*, 14386–14391. IFAC, Toulouse, France.

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.

Wills, A., Mills, A., and Ninness, B. (2011). FPGA implementation of an interior-point solution for linear model predictive control. In *Proceedings of the 18th IFAC World Congress*, 14527–14532. Milano, Italy.

Wills, A.G., Knagge, G., and Ninness, B. (2012). Fast Linear Model Predictive Control Via Custom Integrated Circuit Architecture. *IEEE Transactions on Control Systems Technology*, 20(1), 59–71.

Yang, N., Li, D., Zhang, J., and Xi, Y. (2012). Model predictive controller design and implementation on FPGA with application to motor servo system. *Control Engineering Practice*, 20(11), 1229–1235.

Zhang, P., Zambreno, J., and Jones, P.H. (2017). An Embedded Scalable Linear Model Predictive Hardware-based Controller using ADMM. In *2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. Seattle, WA.