MPC for Deeply Pipelined FPGA Implementation: Algorithms and Circuitry

Juan L. Jerez [#1], Keck-Voon Ling [+2], George A. Constantinides [#3], Eric C. Kerrigan [#*4]

[#] *Department of Electrical and Electronic Engineering, Imperial College London*

London SW7 2AZ, United Kingdom

[+] *School of Electrical and Electronic Engineering, Nanyang Technological University*

Singapore 639798

[*] *Department of Aeronautics, Imperial College London*

London SW7 2AZ, United Kingdom

[1] juan.jerez-fullana@imperial.ac.uk

[2] EKVLING@ntu.edu.sg

[3] george.constantinides@ieee.org

[4] e.kerrigan@imperial.ac.uk

Abstract

Model predictive control (MPC) is an optimization-based scheme that imposes a real-time constraint on computing the solution of a quadratic programming (QP) problem. The implementation of MPC in fast embedded systems presents new technological challenges. In this paper we present a parameterized field-programmable gate array (FPGA) implementation of a customized QP solver for optimal control of linear processes with constraints, which can achieve substantial acceleration over a general purpose microprocessor, especially as the size of the optimization problem grows. The focus is on exploiting the structure and accelerating the computational bottleneck in an existing primal-dual interior-point method. We then introduce a new MPC formulation that can take advantage of the novel computational opportunities, in the form of parallel computational channels, offered by the proposed pipelined architecture to improve performance even further. This highlights the importance of the interaction between the control theory and digital system design communities for the success of MPC in fast embedded systems.

I. Introduction

MPC is an advanced optimal control technology that has proven to be very successful due to its capability of returning an optimal strategy without violating the physical limitations of the system. The need to solve a computationally intensive QP problem at every sampling instant has restricted its applicability to slow plants, such as those encountered in the chemical process industries [1], where sampling times can be on the order of seconds or minutes. As the computational power of new devices continues to rise, MPC is now being proposed for higher bandwidth applications, such as aerospace [2], electrical power generation [3] and automotive [4]–[7]. There is a growing demand for ways of accelerating the solution of QP problems so that the success of MPC can be extended to areas where the computational burden has so far been considered too great.

In terms of online optimization, most attempts at accelerating the solution of QP problems have come in the form of new algorithms or modifications to existing methods with the objective of reducing the computational complexity of the task [8]–[11]. The work presented in this paper differs in that it focuses on the hardware implementation issues of a specific existing algorithm, rather than attempting to modify it.

Recent advances in reconfigurable hardware technology have made the FPGA a suitable platform for scientific computation. FPGAs are a good alternative to application specific integrated circuits (ASICs) for embedded MPC applications since they offer much reduced low-volume cost, greater flexibility, and a shorter design cycle, reducing the risk while still maintaining a high power efficiency. In this work, FPGAs are used as the vehicle to explore the possibilities of parallel hardware, and custom hardware in particular, for the acceleration of quadratic programming solvers for linear MPC.

Section II reviews linear MPC. In Section III the characteristics of existing algorithms for solving QP problems are assessed in terms of their suitability for mapping into hardware. Previous attempts at implementing optimization solvers in custom hardware are reviewed in Section IV. In Section V, the MPC problem is formulated as a QP problem using the approach proposed by Rao *et. al.* [12] and the infeasible primal-dual interior-point method [11] is introduced as our chosen method to solve it. The parallelism opportunities offered by the method, and ways of exploiting the fine and coarse structure in the problem are discussed from a hardware implementation point of view. The FPGA implementation is described in Section VI. Hardware design decisions aiming at exposing all the computational power of the FPGA are explained. In Section VII we present a new MPC formulation based on multiplexed MPC [13] that can take advantage of the new computational opportunities opened by the proposed hardware architecture. The results are presented in Section VIII followed by the conclusion in Section IX.

II. Linear Model Predictive Control

Unlike conventional control techniques, MPC explicitly considers operation on the constraints (saturation) by incorporating the physical limitations of the system into the problem formulation, delivering extra performance gains [1]. However, due to the presence of constraints it is not possible to obtain an analytic expression for the optimum solution and we have to solve an optimization problem at every sample instant, resulting in very high computational demands. In linear MPC, we have a linear model of the plant, linear constraints, and a positive definite quadratic cost function, hence the resulting optimization problem is a convex quadratic program [14]. Without loss of generality, the time-invariant problem can be described by the following equations:

$$\min_{\boldsymbol{u},\boldsymbol{x}} \left[\frac{1}{2} x_T' \widetilde{Q} x_T + \sum_{k=0}^{T-1} \left(\frac{1}{2} x_k' Q x_k + \frac{1}{2} u_k' R u_k + x_k' M u_k \right) \right] \tag{1}$$

subject to:

$$x_0 = \widehat{x} \tag{2a}$$

$$x_{k+1} = A x_k + B u_k \qquad \text{for } k = 0, 1, 2, ..., T-1 \tag{2b}$$

$$J x_k + E u_k \leq d \qquad \text{for } k = 0, 1, 2, ..., T-1 \tag{2c}$$

$$J_T x_T \leq d_T \tag{2d}$$

where $u \in \mathbf{R}^m$, $x \in \mathbf{R}^n$, $\boldsymbol{u} \in \mathbf{R}^{Tm}$ and $\boldsymbol{x} \in \mathbf{R}^{(T+1)n}$ contain the input and state variables at every sampling instant for the whole horizon length T, $\widehat{x}$ is the current estimate of the state of the plant, $Q \in \mathbf{R}^{n\times n}$ is symmetric positive semi-definite (SPSD), $R \in \mathbf{R}^{m\times m}$ is symmetric positive definite (SPD) to guarantee uniqueness of the solution, $M \in \mathbf{R}^{n\times m}$ is such that (1) is convex, $\widetilde{Q} \in \mathbf{R}^{n\times n}$ is an approximation of the cost from $k = T$ to infinity and is SPSD, x' denotes transposition, and $\leq$ denotes componentwise inequality. $A \in \mathbf{R}^{n\times n}$ and $B \in \mathbf{R}^{n\times m}$ are the state transition and control matrices representing the dynamics of the plant, obtained from a discrete-time model or through a suitable discretization of a continuous-time model [15]. $J \in \mathbf{R}^{l\times n}$, $E \in \mathbf{R}^{l\times m}$ and $d \in \mathbf{R}^l$ describe the physical constraints of the system. For instance, upper and lower bounds on the inputs and outputs could be expressed as

$$J := \begin{bmatrix} C \\ -C \\ 0 \\ 0 \end{bmatrix}, E := \begin{bmatrix} 0 \\ 0 \\ I_m \\ -I_m \end{bmatrix}, d := \begin{bmatrix} y_{max} \\ -y_{min} \\ u_{max} \\ -u_{min} \end{bmatrix},$$

$$J_T := \begin{bmatrix} C \\ -C \end{bmatrix}, d_T := \begin{bmatrix} y_{max} \\ -y_{min} \end{bmatrix}, \tag{3}$$

where $C \in \mathbf{R}^{p\times n}$ and p is the number of outputs to the plant.

At every sample instance a measurement of the system's output is taken, from which the current state of the plant is inferred [1]. The optimization problem (1)–(2) is then solved but only the first part of the solution ($u_0^*(\widehat{x})$) is implemented. Due to disturbances, model uncertainties and measurement noise, there will be a mismatch between the next output measurement and what the controller had predicted, hence the whole process has to be repeated again at every sample instant to provide closed-loop stability and robustness.

III. Algorithm Choice

Modern methods for solving QPs can be classified into interior-point or active-set [16] methods, each exhibiting different properties, making them suitable for different puposes. The worst-case complexity of active-set methods increases exponentially with the problem size, and the size of the linear systems that need to be solved at each iteration changes depending on which constraints are active at any given time. In a hardware implementation, this is problematic since all iterations need to be executed on the same fixed architecture. Interior-point methods are a better option for our needs, since they have polynomial complexity and maintain a constant predictable structure, which is easily exploited.

Logarithmic-barrier [14] and primal-dual [17] are two competing interior-point methods. From the implementation point of view, a difference to consider is that the logarithmic-barrier method requires an initial feasible point and fails if an intermediate solution falls outside of the region enclosed by the inequality constraints. In infinite precision this is not a problem, since both logarithmic-barrier and primal-dual methods stay inside the feasible region provided they start inside it. In practice, finite precision effects may lead to infeasible iterates, so in that sense the primal-dual method is more robust. Moreover, in primal-dual there is no need to implement a Phase I method [14] to initialize the algorithm with a feasible point. This issue affects

both hardware and software implementations, although it becomes more critical in hardware since a Phase I method would require explicit support in the form of extra hardware.

Mehrotra's primal-dual algorithm [10] has proven very efficient in software implementations. The algorithm solves two systems of linear equations with the same coefficient matrix in each iteration, thereby reducing the overall number of iterations. However, the benefits can only be attained by using factorization-based methods for solving linear systems, since the factorization is only computed once for both systems. Previous work [18], [19] suggests that iterative methods might be preferable in an FPGA implementation, due to the small number of division operations, which are very expensive in hardware, and because they allow one to trade off accuracy for computation time. In addition, these methods are easy to parallelize since they mostly consist of large matrix-vector multiplications. Furthermore, it is possible to derive pre-conditioners that can help reduce the number of iterations needed to solve the system to the required accuracy [20], and control the ill-conditioning of the matrices towards the later iterations of the interior-point method. As a consequence, simple primal-dual methods, where a single system of equations is solved, could be more suited to the FPGA fabric.

IV. Related Work

Existing work on hardware implementation of optimization solvers can be grouped into those that use interior-point methods [21]–[24] and those that use active-set methods [25], [26]. The suitability of each method for FPGA implementation was studied in [27], highlighting the advantages of interior-point methods for large problems. Occasional numerical instability was also reported, having a greater effect on active-set methods.

An ASIC implementation of explicit MPC [28], based on parametric programming, was described in [29]. The scheme works by dividing the state-space into non-overlapping regions and pre-computing a parametric piecewise linear solution for each region. The online implementation is reduced to identifying the region to which $\widehat{x}$ belongs and implementing a simple linear control law. Explicit MPC is naturally less vulnerable to finite precision effects, and can achieve high performance for small problems, with sampling intervals on the order of μseconds being reported in [29]. However, the memory and computational requirements typically grow exponentially with the problem size, making the scheme unattractive for handling large problems. In this paper we will only consider online numerical optimization, thereby addressing relatively large problems.

The challenge of accelerating linear programs (LPs) on FPGAs was addressed in [22] and [26]. [26] proposed a heavily pipelined architecture based on the simplex method. Speed-ups of around 20x were reported over state-of-the-art LP software solvers, although the method suffers from active-set pathologies when operating on large problems. Acceleration of collision detection in graphics processing was targeted in [22] with an interior-point implementation using single-precision floating point arithmetic, which could operate on relatively small problems.

The feasibility of implementing QP solvers for MPC applications on FPGAs was demonstrated in [21] with a Handel-C implementation exploiting modest levels of parallelism in the interior-point method. The implementation was shown to be able to respond to disturbances and achieve sampling periods comparable to stand-alone Matlab executables for relatively large problems. [24] addressed the implementation of MPC on very resource-constrained embedded systems with an FPGA implementation consisting of a soft-core processor attached to a co-processor used to accelerate computations that allowed data reuse. [23] also proposed a mixed software-hardware implementation where the core matrix computations are implemented

in parallel custom hardware, whereas the remaining operations are implemented on a soft-processor core synthesized into the FPGA. The computational bottlenecks in implementing a logarithmic-barrier method for solving an unstructured QP were identified for determining which computations should be carried out in which unit. However, we will show that if the structure of the QPs arising in MPC is taken into account, we can reach different conclusions as to the location of the computational bottleneck. The hardware implementation of sequential quadratic programming for nonlinear MPC was considered in [25], where the sources of parallelism in an active-set method were identified. The trade-off between data wordlength, computational speed and quality of the applied control was explored in an experimental manner.

V. Primal-dual interior-point algorithm

A. QP Formulation

Following the approach outlined in [12], the optimal control problem (1)–(2) can be written as a sparse QP of the following form:

$$\min_{\theta} \frac{1}{2}\theta^T H \theta$$

$$\text{subject to} \quad F\theta = f$$

$$G\theta \leq g$$

where

$$\theta := [x_0', u_0', x_1', u_1', \ldots, x_{T-1}', u_{T-1}', x_T']'$$

$$H := \begin{bmatrix} \begin{bmatrix} Q & M \\ M' & R \end{bmatrix} \otimes I_T & O \\ O & \widetilde{Q} \end{bmatrix}$$

$$F := \begin{bmatrix} -I_n & & & & \\ A & B & -I_n & & \\ & & \ddots & & \\ & & A & B & -I_n \end{bmatrix} \quad f := \begin{bmatrix} -\widehat{x} \\ O \\ \vdots \\ O \end{bmatrix}$$

$$G := \begin{bmatrix} \begin{bmatrix} J & E \end{bmatrix} \otimes I_T & O \\ O & J_T \end{bmatrix} \quad g := \begin{bmatrix} d \\ \vdots \\ d \\ d_T \end{bmatrix}$$

where $\otimes$ denotes a Kronecker product, I is the identity matrix, and O denotes a matrix or vector of zeros.

In [12], it was shown that by leaving the plant equations as equality constraints in the QP formulation, it was possible to complete the interior-point method in $\mathcal{O}(T)$ operations as opposed to $\mathcal{O}(T^3)$ required with the dense formulation. In our current implementation, the latency is $\mathcal{O}(T^2)$ because the number of iterations of the linear solver depends on T, and we have not

completely exploited a full parallelization of the solver. Future work will investigate the implementation of a pre-conditioner [20] and further use of parallelization to further reduce the latency.

In terms of memory requirements, the coefficient matrix $\mathcal{A}_k$, as shown in Algorithm 1, requires storage space for approximately $\frac{1}{2}(Tm)^2$ elements when using the dense formulation and approximately $T(2n+m)^2$ non-zero elements using the sparse formulation (considering symmetry in both cases). For problems with large horizon lengths, a method that only stores non-zero elements would provide an important memory saving when employing the sparse formulation. In Section VI we will introduce a storage method that exploits the fine grain structure in $\mathcal{A}_k$ to allow us to store significantly fewer elements than $T(2n+m)^2$.

B. Algorithm Description

The primal-dual algorithm uses Newton's method [14] for solving a nonlinear system of equations, known as the Karush-Kuhn-Tucker (KKT) optimality conditions. The method solves a sequence of related linear problems. At each iteration, three tasks need to be performed: linearization around the current point, solving the resulting linear system to obtain a search direction, and performing a line search to update the solution to a new point. In this work we use the infeasible primal-dual interior-point method introduced in [11]. For completeness, we include a brief description of the method.

The Lagrangian function for this optimization problem is given by

$$L(\theta,\nu,\lambda,s) := \frac{1}{2}\theta^T H\theta + \nu'(F\theta - f) + \lambda'(G\theta - g + s),$$

where ν and λ are known as the Lagrange multipliers and s is a vector of slack variables. Minimization of the Lagrangian gives rise to the KKT conditions, where the last equation is known as the complementary condition:

$$\begin{aligned} H\theta + F^T\nu + G^T\lambda &= 0, && \text{(4a)}\\ F\theta - f &= 0, && \text{(4b)}\\ G\theta - g + s &= 0, && \text{(4c)}\\ \Lambda S e &= 0,\ \lambda, s \geq 0, && \text{(4d)} \end{aligned}$$

where Λ and S are diagonal matrices with non-zero elements taken from λ and s, respectively, and e is a vector of ones. Linearization of (4) gives rise to the following linear system (the subscript k refers to the iteration number in the interior-point method)

$$\begin{bmatrix} H & F^T & G^T & 0 \\ F & 0 & 0 & 0 \\ G & 0 & 0 & I \\ 0 & 0 & S_k & \Lambda_k \end{bmatrix} \begin{bmatrix} \Delta\theta_k \\ \Delta\nu_k \\ \Delta\lambda_k \\ \Delta s_k \end{bmatrix} = \begin{bmatrix} r_k^H \\ r_k^F \\ r_k^G \\ r_k^S \end{bmatrix}, \qquad (5)$$

where

$$r_k^H := -H\theta_k - F'\nu_k - G'\lambda_k, \tag{6a}$$

$$r_k^F := -F\theta_k + f, \tag{6b}$$

$$r_k^G := -G\theta_k + g - s_k, \tag{6c}$$

$$r_k^S := -\Lambda_k S_k e + \sigma\mu_k e, \tag{6d}$$

and

$$\mu_k := \frac{\lambda_k' s_k}{Tl + 2p}. \tag{7}$$

σ is a small number between zero and one known as the centrality parameter, which is included to make sure that the progress of the method does not stop when it hits the boundaries of the feasible region [17]. Note that the solution to this linear system is a modified Newton direction (modification in (6d)).

Block elimination can be applied twice to (5) to reduce the size of the system without destroying the structure in the problem. The resulting linear system is

$$\begin{bmatrix} H + G'W_kG & F' \\ F & 0 \end{bmatrix} \begin{bmatrix} \Delta\theta_k \\ \Delta\nu_k \end{bmatrix} = \begin{bmatrix} r_k^\theta \\ r_k^\nu \end{bmatrix}, \tag{8}$$

where

$$W_k := \Lambda_k S_k^{-1}, \tag{9}$$

$$r_k^\theta := r_k^H + G'W_k(r_k^G + s_k - \sigma\mu_k\Lambda_k^{-1}e), \tag{10}$$

$$r_k^\nu := r_k^F, \tag{11}$$

and

$$\Delta\lambda_k = -W_k(r_k^G + s_k - \sigma\mu_k\Lambda_k^{-1}e - G\Delta\theta_k), \tag{12}$$

$$\Delta s_k = -s_k - W_k^{-1}\Delta\lambda_k + \sigma\mu_k\Lambda_k^{-1}e. \tag{13}$$

Algorithm 1 summarizes this process. The number of iterations (I_{NW}) depends on the required accuracy of the solution but not on the size of the MPC problem. The infeasible primal-dual method can be initialised with any arbitrary point $(\theta_0, \nu_0, \lambda_0, s_0)$ satisfying $[\lambda_0', s_0']' > 0$. Several authors have proposed a warm-start method [8] where the current problem is initialised with a shifted version of the solution at the previous sampling instant, based on the observation that in the absence of large disturbances the solution to adjacent QPs is very similar. In the FPGA implementation presented in this paper, the method is initialised with the solution from the previous problem without shifting. This avoids movement of data between memory locations.

C. Algorithm analysis

When calculating the Hessian $\mathcal{A}_k$, only the diagonal matrix W_k is changing from iteration to iteration, so only

$$G'W_kG = \begin{bmatrix} I_T \otimes \begin{bmatrix} J'\mathcal{W}_iJ & J'\mathcal{W}_iE \\ E'\mathcal{W}_iJ & E'\mathcal{W}_iE \end{bmatrix} & O \\ O & J_T'\mathcal{W}_TJ_T \end{bmatrix}$$

Algorithm 1 QP algorithm

Choose initial point $(\theta_0, \nu_0, \lambda_0, s_0)$ with $[\lambda_0', s_0']' > 0$

for $k = 0$ to $I_{NW} - 1$ **do**

1. $\mathcal{A}_k := \begin{bmatrix} H + G'W_kG & F' \\ F & 0 \end{bmatrix}$
2. $b_k := \begin{bmatrix} r_k^\theta \\ r_k^\nu \end{bmatrix}$
3. Solve $\mathcal{A}_k z_k = b_k$ for $z_k =: \begin{bmatrix} \Delta\theta_k \\ \Delta\nu_k \end{bmatrix}$
4. Compute $\Delta\lambda_k$
5. Compute Δs_k
6. Find $\alpha_k := \max_{(0,1]} \alpha : \begin{bmatrix} \lambda_k + \alpha\Delta\lambda_k \\ s_k + \alpha\Delta s_k \end{bmatrix} > 0.$
7. $(\theta_{k+1}, \nu_{k+1}, \lambda_{k+1}, s_{k+1}) :=$
   $(\theta_k, \nu_k, \lambda_k, s_k) + \alpha_k(\Delta\theta_k, \Delta\nu_k, \Delta\lambda_k, \Delta s_k)$

end for

needs to be computed, where $\mathcal{W}_i$ are diagonal $l \times l$ submatrices of W_k. If the constraints are separable in state and input constraints, $J'\mathcal{W}_iE$ and $E'\mathcal{W}_iJ$ are zero. In addition, J and E are usually sparse. A common situation is having upper and lower bounds on the inputs and outputs of the system (as described by (3)). In this case, computing the matrix triple product $E'\mathcal{W}_iE$ consists of $2m$ additions instead of $\Omega(ml^2 + m^2l)$ operations. Similarly, $J'\mathcal{W}_iJ$ consists of two small matrix row updates plus two small matrix-matrix multiplications. The coarser structure of H, F and G can also be used when calculating r_k^θ, r_k^ν, $\Delta\lambda_k$ and Δs_k. This results in having to compute many small matrix-vector multiplications in standard and transposed form.

Exploiting the finer structure in a software implementation would be inefficient as it would involve complex array index arithmetic resulting in non-coherent memory reads. In a CPU, this will lead to an increased number of cache misses. Moreover, having to perform many small matrix-vector multiplications means that there will be many movements of small blocks of data from main memory to the processor cache resulting in significant time penalties. However, in custom hardware there is a flexible memory subsystem that can be designed such that data is always available when and where is it needed, improving data locality and fully avoiding cache misses. Furthermore, if appropriate support is provided, there is no difference whether we access matrix data by row or by column, hence standard and transposed multiplications are equally efficient.

When solving $\mathcal{A}_k z_k = b_k$ using an iterative method, most of the computations are associated with computing a large matrix-vector product. This kind of computation can be carried out efficiently in a microprocessor, especially if the whole matrix can be

accommodated inside the processor cache, as there will be next to no main memory accesses. In addition, some microprocessors include explicit support for carrying out a multiply-accumulate instruction in one cycle. However, sequential software cannot take advantage of the easy parallelization opportunities available for this computation. A graphical processing units' (GPU) instruction set is a good match for single-instruction multiple-data (SIMD) computations, although the lack of independence between additions in a dot-product calculation limits the speed-up achievable with a GPU architecture. The FPGA's flexibility allows us to create a custom datapath to best exploit the dataflow in a computation, allowing wider parallelization and deep pipelining. An example of a customized architecture for computing dot-products is shown in Figure 1.

VI. Implementation

A. Linear Solver

Most of the computational complexity in each iteration of the interior-point method is associated with solving the system of linear equations $\mathcal{A}_k z_k = b_k$, hence it makes sense to concentrate our efforts in accelerating this task. After appropriate row re-ordering (interleaving elements of θ and ν), the indefinite symmetric matrix $\mathcal{A}_k$ becomes banded (refer to Figure 4 for more details). The order (number of rows or columns) and half-bandwidth of $\mathcal{A}_k$ in terms of the control problem parameters are given respectively by

$$N := T(2n+m)+2n, \tag{14a}$$

$$M := 2n+m. \tag{14b}$$

Notice that the number of outputs p and the number of constraints l does not affect the order or bandwidth of $\mathcal{A}_k$. As a consequence, these parameters have no effect on the complexity of the linear solver, which we will show to determine the overall computation time.

The minimum residual (MINRES) method is a suitable iterative algorithm for solving linear systems with indefinite symmetric matrices [30]. At each MINRES iteration, a matrix-vector multiplication accounts for the majority of the computations. This kind of operation is easy to parallelize and consists of multiply-accumulate instructions, which are known to map efficiently into hardware in terms of resources.

In [19], the authors propose an FPGA implementation of the MINRES method, reporting speed-ups of around one order of magnitude over software implementations. Most of the acceleration is achieved through a deeply pipelined dedicated hardware block (shown in Figure 1) that parallelizes dot-product operations for computing the matrix-vector multiplication in a row-by-row fashion. Notice that the size of the dot-products that are computed in parallel is independent of the control horizon length T (refer to (14b)), thus we do not expect the computational resource usage to scale with T. However, the number of elements in $\mathcal{A}_k$ that need to be stored does scale as $\mathcal{O}(T)$ and the compute time is $\mathcal{O}(T^2)$ when taking N MINRES iterations.

Since the dot-product unit is deeply pipelined it can accept new data at every clock cycle, thus a whole matrix-vector multiplication will take N cycles to be introduced into the dot-product block, and it should be possible to complete a MINRES iteration every N cycles if the remaining operations are computed using other units. However, the latency of one iteration is greater than this due to the depth of the adder tree and the other operations in the MINRES iteration. This throughput-latency mismatch can be used to our advantage processing several independent problems simultaneously to make sure the dot-product

hardware is active at all times. The number of problems that can be processed simultaneously in the linear solver in terms of the matrix dimensions is given by

$$P := \left\lceil \frac{3N + k_1\lceil \log_2(2M-1)\rceil + k_2}{N} \right\rceil, \tag{15}$$

where $k_1 = 24$ and $k_2 = 154$ in the current implementation. The linear term results from the row by row processing for the matrix-vector multiplication and serial-to-parallel conversions, whereas the log term comes from the depth of the adder reduction tree in the dot-product block. The constant term comes from the other operations in the MINRES iteration. The latency for one MINRES iteration is given by $PN + P$ since the dot-product block is being reused to compute a 2-norm for each processed problem.

Using this approach, $\Theta(M)$ resources are being used to reduce the latency of one iteration from $\mathcal{O}(MN)$ to $\mathcal{O}(N)$. As device density increases, it will become possible to completely parallelise matrix-vector multiplications by having N dot-product modules operating in parallel to reduce the latency even further. The new expression for P would be

$$P := \left\lceil \frac{k_1\lceil \log_2(2M-1)\rceil + k_2}{1} \right\rceil, \tag{16}$$

where the value of k_1 and k_2 would depend on the implementation. In a MINRES FPGA implementations, where the matrix data has to be fed from outside the chip, the increase in I/O bandwidth requirements imposed by the increase in parallelism will quickly exceed the capabilities of standard interfaces, such as PCI Express. However, in our QP solver implementation, matrix data is generated on-chip, hence the I/O requirements are very low (refer to Part E), and the increase in average I/O requirements will still be within the capabilities of PCI Express. Notice that the latency of the MINRES implementation will stop depending on the horizon length T. However, for the same control parameters, more independent problems would be needed to fill the pipeline, as a consequence of the architecture being able to compute one matrix-vector multiplication every cycle.

B. Pipelining

The remaining operations in the interior-point iteration are undertaken by a separate hardware block, which we call Stage 1. The resulting two-stage architecture is shown in Figure 2.

Since the linear solver will provide most of the acceleration and consume most resources it is vital that it remains busy at all times. The whole design can be seen as a high-level pipeline with two stages where the computational times have to be matched to achieve the highest hardware efficiency.

Notice that if both blocks are to be doing useful work at all times, while the linear system for a specific problem is being solved, Stage 1 has to be updating the solution and linearizing for another independent problem. In addition, the architecture described in [19] can process P independent problems simultaneously, so our design can process $2P$ independent QP problems simultaneously (as a result of the high level pipeline) at no extra hardware cost. Figure 3 shows the number of available parallel computation channels for problems with different parameters. It is important to note that P converges to a small number ($P = 4$) as the size of A_k increases, thus even for relatively large problems there are eight independent threads available for further exploitation in our proposed hardware.

Another approach could have been to reuse the linear solver hardware to perform the operations in Stage 1, however, the hardware would be used inefficiently and the reduction in latency would likely be small if any. We believe our approach to be appropriate, as it allows us to expose all the computational power of the device and open up new possibilities for algorithm development. In Section VII we present a new method that can take advantage of the independent parallel computational channels offered by our proposed architecture.

C. Architecture for Sequential Block

When computing the coefficient matrix $\mathcal{A}_k$, only the diagonal matrix W_k changes from one iteration to the next, and the constraint matrices J and E are generally sparse, thus the complexity of this calculation is relatively small. If the structure of the problem is taken into account, we find that the remaining calculations in an interior-point iteration are all sparse and very simple compared to the linear solver. Comparing the computational count of all the operations to be carried out in Stage 1 with the latency of the linear solver implementation given by

$$(PN + P)I_{MR}, \tag{17}$$

we come to the conclusion that for most control problems, the optimum implementation of Stage 1 is sequential, as this will be enough to keep the linear solver busy at all times (I_{MR} is the number of iterations the MINRES method takes to solve the linear system to the required accuracy). This is a consequence of the latency of the linear solver being $\mathcal{O}(T^2)$ [19], whereas the complexity of Stage 1 is only $\mathcal{O}(T)$. If this is not the case, it is possible to have several instances of Stage 1 running in parallel for the different independent problems that are being processed simultaneously. This will result in a small increase in computational resources (refer to Table I) and a negligible increase in memory requirements as the control block will be shared by all parallel instances. However, we have observed that this situation only occurs for very small problems and usually two parallel blocks are enough to solve the problem when it arises.

1) Datapath: The computational block performs any of the main arithmetic operations: addition/subtraction, multiplication and division. Xilinx Core Generator [31] was used to generate highly optimized single-precision floating point units with maximum latency to keep pipeline stages short and achieve a high clock frequency. Extra registers were added after the multiplier to match the latency of the adder for synchronization, as these are the most common operations. The latency of the divider was much larger (27 cycles) than the adder (12 cycles), therefore it was decided not to match the delay of the divider path, as it would reduce our flexibility for ordering computations. A small number of no operations (NOPs) were inserted (to account for the latency mismatch) whenever division operations were needed, namely only when calculating W_k.

Comparison operations are also required for the line search method (Line 6 of Algorithm 1), however this is implemented by repeated comparison with zero, so only the sign bit needs to be checked and a full floating-point comparator is not required.

Table I shows the total number of floating point units in the circuit, which account for most of the computational resources required by the design. We can see that the resources consumed by Stage 1 are a very small fraction of the total, so the impact of having several copies of Stage 1 running in parallel is also small.

TABLE I: Total number of floating point units in the circuit in terms of the bandwidth of $\mathcal{A}_k$ and the parameters of the control problem. This is independent of the horizon length T. i is the number of parallel instances of Stage 1, which is 1 for most problems.

	Floating point units	
	Matrix parameters	Control parameters
Stage 1	$3i$	$3i$
Dot-product (linear solver)	$4M-3$	$8n+4m-3$
Other (linear solver)	27	27
Total	$4M+24+3i$	$8n+4m+24+3i$

2) Control block: Since the same computational units are being reused to perform many different operations, the necessary control is rather complex. The control block needs to provide the correct sequence of read and write addresses for the data RAMs, as well as other control signals, such as computation selection. An option would be to store the values for all control signals at every cycle in a program memory and have a counter iterating through them. However, this would take a large amount of memory. For this reason it was decided to trade a small increase in computational resources for a much larger decrease in memory requirements.

Frequently occurring memory access patterns have been identified and a dedicated address generator hardware block has been built to generate them from minimum storage. Each pattern is associated with a control instruction. Examples of these patterns are: simple increments $a, a+1, ..., a+b$ and the more complicated read patterns needed for matrix vector multiplication (standard and transposed). This approach allows storing only one instruction for a whole matrix-vector multiplication or for an arbitrary long sequence of additions. Control instructions to perform line search and linearization for one problem were stored. When the last instruction is reached, the counter goes back to instruction 0 and iterates again for the next problem with the necessary offsets being added to the control signals.

3) Memory subsystem: Separate memory blocks were used for data and control instructions, allowing simultaneous access and different word-lengths in a similar way to a Harvard microprocessor architecture. However, in our circuit there are no cache misses and a useful result can be produced almost every cycle. The data memories are divided in two blocks, each one feeding one input of the computational block. The intermediate results can be stored in any of these simple dual-port RAMs for flexibility in ordering computations. The memory to store the control instructions is divided into four single port ROMs corresponding to read and write addresses of each of the data RAMs. The responsibility for generating the remaining control signals is spread out over the four blocks.

Another approach for implementing Stage 1 could have been using an off-chip microprocessor, however, the data for matrix $\mathcal{A}_k$ would need to be transmitted from the CPU to the FPGA, and as a result I/O will determine how much parallelism can be employed in the linear solver. A further alternative would be using an on-chip soft-core processor. In this case, the lesser amount of customization compared to our implementation would result in increased storage requirements for instructions [24].

In addition, a soft-core processor could lower the operating frequency of the design.

D. Latency and throughput

Since the FPGA has deterministic timing, we can calculate the exact latency and throughput of our system. The overall latency of the circuit will be given by

$$\text{Latency} = \frac{2I_{NW}(PN+P)I_{MR}}{\text{FPGA}_{freq}} \text{ seconds,} \tag{18}$$

where I_{NW} is the number of outer iterations in the interior-point method (Algorithm 1), FPGA_{freq} is the FPGA's clock frequency, and P is given by (15). In that time the controller will be able to output the result to $2P$ problems.

E. Input/Output

Stage 1 is responsible for handling the chip I/O. It reads the current state measurement $\widehat{x}$ as n 32-bit floating point values sequentially through a 32-bit parallel input data port. Outputting the m 32-bit values for the optimal control move $u_0^*(\widehat{x})$ is handled in a similar fashion. When processing $2P$ problems, the average I/O requirements are given by

$$\frac{2P(32(n+m))}{\text{Latency given by (18)}} \text{ bits/second.}$$

For the example problems that we have considered in Section VIII, the I/O requirements range from 0.2 to 0.00015 Mbits/second, which is well within any standard FPGA platform interface, such as PCI Express. The combination of a very computationally intensive task with very low I/O requirements, highlight the affinity of the FPGA for MPC computation.

F. Coefficient Matrix Storage

When implementing an algorithm in software, a large amount of memory is available for storing intermediate results. In FPGAs, there is a very limited amount of fast on-chip memory, around 4.5Mbytes for high-end memory-dense Xilinx Virtex-6 devices [32]. If a particular design requires more memory than available on chip, there are two negative consequences. Firstly, if the size of the problems we can process is limited by memory, it means that the computational capabilities of the device are not being fully exploited, since there will be unutilized slices and digital signal processing (DSP) blocks. Secondly, if we were to try to overcome this problem by using off-chip memory, the performance of our circuit is likely to suffer since off-chip memory accesses are slow compared to the on-chip clock frequency. By taking into account the special structure of the matrices that are fed to the linear solver in the context of MPC, we can substantially reduce memory requirements so that this issue affects a smaller group of problems.

The matrix $\mathcal{A}_k$ is banded and symmetric (after re-ordering). On-chip buffering of this type of matrix using compressed diagonal storage (CDS) can achieve substantial memory savings with minimum control overhead in an FPGA implementation of the MINRES method [33]. The memory reductions are achieved by only storing the non-zero diagonals of the original matrix as columns of the new compressed matrix. Since the matrix is also symmetric, only half of the CDS matrix needs to be stored. In order to achieve the same result when multiplying by a vector, the vector has to be aligned with its corresponding matrix components. It turns out that this is simply achieved by shifting the vector by one position at every clock cycle.

The method described in [33] assumes a dense band; however, it is possible to achieve further memory savings by exploiting the structure of the MPC problem even further. The structure of the original matrix and corresponding CDS matrix for a small MPC problem are shown in Figure 4, showing variables (elements that can vary from iteration to iteration of the interior-point method) and constants. The following observations can be exploited to store much fewer elements than with standard CDS:

- Non-zero blocks are separated by layers of zeros in the CDS matrix. It is possible to store only one of these zeros per column and add common circuitry to generate appropriate sequences of read addresses.
- Only a few diagonals adjacent to the main diagonal in $\mathcal{A}_k$ have varying elements. This means that only few columns in the CDS matrix contain varying elements. This has important implications, since in the original linear solver implementation [19], matrices for the P problems that are being processed simultaneously have to be buffered on-chip. The memory blocks holding columns of the matrix (refer to Figure 1) have to be double in size to allow writing the data for the next P problems while reading the data for solving the current P problems. Constant columns in the CDS matrix are common for all problems, hence the memories used to store them can be much smaller.
- Constant columns consist of repeated blocks of size $2n + m$ (where n values are zeros or ones), hence further memory savings can be attained by only storing one of these blocks per column.

A memory controller for the variable columns and another memory controller for the constant columns were created in order to be able to generate the necessary memory access patterns to implement this reduced storage scheme. The impact upon the overall performance is negligible, since these controllers consume few resources compared with floating point units and they do not slow down the circuit. Using this storage technique in a software implementation would require cumbersome array index arithmetic, which will likely lead to performance degradation.

If we consider a dense band, storing the coefficient matrix using CDS would require $2P(T(2n+m)+2n)(2n+m)$ elements. By taking into account the sparsity of matrices arising in MPC, it is possible to only store $2P(1 + T(m + n) + n)n + (1 + m + n)(m + n)$ elements. Figure 5 compares the memory requirements for storing the coefficient matrices on-chip when considering: a dense matrix, a banded symmetric matrix and an MPC matrix (all in single-precision floating-point). Memory savings of approximately 75% can be achieved by considering the in-band structure of the MPC problem compared to the standard CDS implementation. In practice, columns are stored in BlockRAMs of discrete sizes, therefore actual savings will differ slightly in an FPGA implementation.

VII. Parallel Multiplexed MPC

Multiplexed MPC (MMPC) has been proposed elsewhere [13], [34]. This section extends the MMPC algorithm in a way such that the architecture proposed in Section VI, which is capable of solving $2P$ QP problems in parallel can be exploited. We called this version of MMPC, Parallel MMPC.

The original formulation of MMPC was for implementation on a single core processor, solving one QP problem per sampling interval. The key idea in MMPC is that, for an m-input plant, instead of optimizing over all the m input channels in one large QP, the inputs are optimized one channel at a time, in a pre-planned periodic sequence, and the control moves updated as soon as the solution becomes available. This results in a smaller QP at each sampling instant, hence reduced online computational

load which in turn enables faster sampling, leading to faster response to disturbances, despite finding a sub-optimal solution to the original optimization problem [35].

MMPC is closer to industrial practice in cases where there is a complex plant with network constraints, meaning that all control inputs cannot be updated simultaneously due to limitations in the communication channels between the actuators and the controller. Parallel MMPC helps to choose which inputs are best to update at any given sampling interval.

A detailed derivation of the Parallel MMPC is beyond the scope of this paper. Instead, we set out the following algorithm which outlines the key steps in Parallel MMPC:

Algorithm 2 Parallel MMPC

1. Initialize by optimizing over all the control moves.
2. Stored planned moves (T moves for each input).

while 1 **do**

3. Apply the first control move for all inputs and shift the plan.
4. Obtain new measurement $\widehat{x}$.
5. Solve m different copies of MMPC in parallel. For each copy, optimize with respect to different subsets of control moves.
6. Evaluate and select from these m copies of MMPC, the set of control moves that gives the smallest cost.
7. Update the plan for the set of control moves that gives the smallest cost.

end while

As can be seen from Algorithm 2, Parallel MMPC uses MMPC as an elementary building block. In Parallel MMPC, for a plant with m inputs, at a given time, there can be up to m copies of MMPC. Each of these operates independently and in parallel, and when given the current plant state $\widehat{x}$, optimized with respect to different subsets of control moves. The set of control moves which produces the smallest cost is selected and applied to the plant. The process is repeated at the next updating instant. The resulting updating sequence does not follow a pre-planed sequence and is not necessarily periodic.

Note that Step 1 of Algorithm 2 involves solving for inputs across all input channels. This type of initialization requirement is common in distributed MPC. Subsequent optimizations use this initial solution, but optimize with respect to a subset of control moves. The stability property of MMPC does not depend on the optimality of this initial solution, only its feasibility [36].

Proposition 1: Parallel MMPC, obtained by implementing Algorithm 2, gives closed-loop stability.

A detailed proof is beyond the scope of this paper and only an outline is provided. The proof follows standard argument used by most MPC stability proofs. It depends on the constrained optimization being feasible at each step. In the proposed Parallel MMPC algorithm, the default MMPC is always evaluated at every iteration, among the m parallel copies of MMPC. It then follows that closed-loop stability can be achieved by applying the default MMPC, which is stabilizing. This gives the worst case since the Parallel MMPC algorithm ensures that switching to a different MMPC will further reduce the cost.

VIII. RESULTS

Part A is valid for both conventional and multiplexed MPC, whereas parts B and C are based on conventional MPC results and part D presents the improvement of parallel MMPC.

A. Resource Usage

An FPGA consists of a 2D array of configurable logic blocks (CLBs), which contain look-up tables (LUTs) and registers to implement logic functions, embedded configurable multiplier blocks and embedded configurable memories. Table II summarises the dependence of the different resources on the control parameters.

TABLE II: Resource dependence on the different control parameters. The computational resources represent LUTs, registers and embedded multipliers. We assume that the problems are big enough for $P = 4$, hence the expression for P does not affect the memory requirements.

	Computational Resources	Memory Resources
Number of inputs	$\Theta(m)$	$\mathcal{O}(m^2)$
Number of states	$\Theta(n)$	$\mathcal{O}(n^2)$
Horizon length	$\Theta(1)$	$\mathcal{O}(T)$

The proposed design was synthesized using Xilinx XST inside Xilinx ISE 12 targeting a Virtex 6 VSX 475T [32]. Figure 6 illustrates how the different resources scale with the number of states confirming the dependencies stablished in Table II. The jumps in the memory requirements curve originate when the number of elements to be stored in the RAMs holding variable columns exceeds the size of Xilinx BlockRAMs, which come in discrete sizes.

B. Latency vs Absolute Horizon Length

For a fixed horizon length in seconds T_c, sampling faster means that the optimization problem will become larger (as the horizon length in terms of steps T will be larger), resulting in longer computational times. Figure 7 explores this relationship. The critical sampling line represents the point where the sampling interval equals the computational time. For conventional MPC, the operating point has to be in the right-half side of the graph. Operation on the left-half part where the sampling frequency is greater than the computational delay will be investigated in the future in the context of a pipelined computing architecture.

The computational time given by (18) is $\mathcal{O}(T^2)$, whereas the sampling interval is given by

$$T_s := \frac{T_c}{T}, \tag{19}$$

hence at the point of critical sampling T is $\mathcal{O}(\sqrt[3]{T_c})$. This means that the computational time will be $\mathcal{O}(T_c^{\frac{2}{3}})$, which is what we observe in Figure 8. The plot is generated by taking the operating point immediately to the right of the critical sampling line for many different curves. This was repeated for systems with different number of states.

C. Performance of Classical MPC

Post place-and-route results showed that a clock frequency above 150MHz is achievable with very small variations for different problem sizes, since the critical path is inside the control block in Stage 1. Figure 9 shows the latency and throughput performance of the FPGA and latency results for a microprocessor implementation. For the software benchmark, we have used a direct C sequential implementation, compiled using GCC -O4 optimizations running on a Intel Core2 P7350 with 3GB of RAM, 3MB L2 cache, and a clock frequency of 2GHz. Note that for matrix operations of this size, this approach produces better performance software than using libraries such as Intel MKL.

The FPGA implementation starts to outperform the microprocessor as soon as there is enough parallelism to overcome the clock frequency disadvantage (this happens when $n > 3$ for the example problem). The performance gap widens as the size of the optimization problem increases as a result of increased parallelism in the linear solver. The FPGA throughput curve represents the number of interior-point iterations per second when multiplexing $2P$ independent problems into the device to fill the pipeline.

The normalized CPU curve in Figure 9 illustrates the performance of a sequential implementation running at the same frequency as the FPGA, hence can be used to compare the number of cycles needed in both implementations. Since the power consumption of the CPU will be larger than the FPGA as a result of a higher operating frequency, this curve can also be used as an indication of the power efficiency of both implementations. However, the actual power consumption of the two implementations is hard to estimate.

For large problems, comparing against an efficient microprocesor implementation of the same algorithm, the current FPGA implementation can provide close to one order of magnitude reduction in latency and two orders of magnitude improvement in throughput if there are enough problems to fill the pipeline. In terms of clock cycles, there will be an extra order of magnitude performance improvement.

D. Performance of Multiplexed MPC

Figure 10 compares the computational times for standard MPC and multiplexed MPC when taking advantage of the parallel computational channels provided by the architecture proposed in Section VI.

Systems with a larger number of inputs will benefit most from employing the multiplexed MPC formulation, as the reduction in size of the QP problem will be larger. Expression (18) consists of quadratic, linear and constant terms with respect to the number of inputs. If m is small compared to n and T, the constant term dominates and the improvement from using multiplexed MPC diminishes as a consequence. When m is large relative to n and T, the quadratic and linear terms gain more weight, hence the improvement becomes very significant.

IX. Conclusion

This paper has described a parameterizable FPGA architecture for solving QP optimization problems in linear MPC. Various design decisions have been justified based on the significant exploitable structure in the problem. The main source of acceleration is a parallel linear solver block, which reduces the latency of the main computational bottleneck in the optimization method. Results show that a significant reduction in latency is possible compared to a sequential software implementation, which

translates to high sampling frequencies and better quality control. We have presented a new MPC formulation that breaks the original problem into smaller subproblems in order to exploit the high throughput of the proposed FPGA architecture.

We are currently working on completely automating the design flow with the objective of making efficient use of all resources available in any given target FPGA platform, avoiding the situation observed in Figure 6, where a large proportion of the logic resources remain unutilized. The potential for industrial take-up of this technology is currently being explored with our partners.

ACKNOWLEDGMENT

The authors would like to acknowledge the support of the EPSRC (Grant EP/G031576/1), discussions with Prof. Jan Maciejowski, Mr. David Boland and Mr. Amir Shazhad, and industrial support from Xilinx, the Mathworks, and the European Space Agency.

REFERENCES

[1] J. Maciejowski, *Predictive Control with Constraints*. Harlow, UK: Pearson Education, 2001.

[2] T. Keviczky and G. J. Balas, "Receding horizon control of an F-16 aircraft: A comparative study," *Control Engineering Practice*, vol. 14, no. 9, pp. 1023–1033, Sep 2006.

[3] G. Prasad, G. W. Irwin, E. Swidenbank, and B. W. Hogg, "Plant-wide predictive control for a thermal power plant based on a physical plant model," *IEE Proc. Control Theory and Applications*, vol. 147, no. 5, pp. 523–537, Sep 2000.

[4] H. J. Ferreau, P. Ortner, P. Langthaler, L. del Re, and M. Diehl, "Predictive control of a real-world diesel engine using an extended online active set strategy," *Annual Reviews in Control*, vol. 31, p. 293301, 2007.

[5] M. Morari, M. Baotic, and F. Borrelli, "Hybrid systems modeling and control," *European Journal of Control*, vol. 9, no. 2-3, pp. 177–189, Apr 2003.

[6] A. Vahidi, A. G. Stefanopoulou, and H. Peng, "Model predictive control for starvation prevention in a hybrid fuel cell system," in *Proc. American Control Conference*, Boston, MA, USA, Jun 2004, pp. 834–839.

[7] L. D. Re, L. Glielmo, C. Guardiola, and I. Kolmanovsky, Eds., *Automotive Model Predictive Control: Models, Methods and Applications*, 1st ed., ser. Lecture Notes in Control and Information Sciences. Springer, 2010.

[8] Y. Wang and S. P. Boyd, "Fast model predictive control using online optimization," *IEEE Trans. Control Syst. Technol.*, vol. 18, no. 2, pp. 267–278, Mar 2010.

[9] S. G. Nash, "A survey of truncated Newton methods," *Journal of Computational and Applied Mathematics*, vol. 124, no. 1-2, pp. 45–59, Dec 2000.

[10] S. Mehrotra, "On the implementation of a primal-dual interior point method," *SIAM Journal on Optimization*, vol. 2, no. 4, pp. 575–601, Nov 1992.

[11] S. J. Wright, "Applying new optimization algorithms to model predictive control," in *Proc. Int. Conf. Chemical Process Control*, Tahoe City, CA, USA, Jan 1996, pp. 147–155.

[12] C. V. Rao, S. J. Wright, and J. B. Rawlings, "Application of interior-point methods to model predictive control," *Journal of Optimization Theory and Applications*, vol. 99, no. 3, pp. 723–757, Dec 1998.

[13] K.-V. Ling, J. M. Maciejowski, and B. F. Wu, "Multiplexed model predictive control," in *Proc. 16th IFAC World Congress*, Prague, Czech Republic, July 2005.

[14] S. P. Boyd and L. Vandenberghe, *Convex Optimization*. Cambridge, UK: Cambridge University Press, 2004.

[15] G. F. Franklin, J. D. Powell, and M. L. Workman, *Digital Control of Dynamic Systems*, 3rd ed. Prentice Hall, 1997.

[16] J. Nocedal and S. J. Wright, *Numerical Optimization*. New York, USA: Springer, 2006.

[17] S. J. Wright, *Primal-Dual Interior-Point Methods*. Philadelphia, USA: SIAM, 1997.

[18] A. R. Lopes and G. A. Constantinides, "A high throughput FPGA-based floating-point conjugate gradient implementation," in *Proc. 4th Int. Workshop on Applied Reconfigurable Computing*, London, UK, Mar 2008, pp. 75–86.

[19] D. Boland and G. A. Constantinides, "An FPGA-based implementation of the MINRES algorithm," in *Proc. Int. Conf. on Field Programmable Logic and Applications*, Heidelberg, Germany, Sep 2008, pp. 379–384.

[20] A. Shahzad, E. C. Kerrigan, and G. A. Constantinides, "A fast well-conditioned interior point method for predictive control," in *Proc. 49th IEEE Conf. on Decision and Control*, Atlanta, Georgia, USA, Dec 2010, (Accepted).

[21] K.-V. Ling, B. F. Wu, and J. M. Maciejowski, "Embedded model predictive control (MPC) using a FPGA," in *Proc. 17th IFAC World Congress*, Seoul, Korea, Jul 2008, pp. 15 250–15 255.

[22] C.-H. Wu, S. O. Memik, and S. Mehrotra, "FPGA implementation of the interior-point algorithm with application to collision detection," in *Proc. 17th IEEE Symp. on Field Programmable Custom Computing Machines*, Napa, CA, USA, Apr 2009, pp. 295 –298.

[23] L. G. Bleris, P. D. Vouzis, M. G. Arnold, and M. V. Kothare, "A co-processor FPGA platform for the implementation of real-time model predictive control," in *Proc. American Control Conference*, Minneapolis, USA, Jun 2006, pp. 1912–1917.

[24] S. L. Koh, "Solving interior point method on a FPGA," Master's thesis, Nanyang Technological University, Singapore, 2009.

[25] G. Knagge, A. Wills, A. Mills, and B. Ninnes, "ASIC and FPGA implementation strategies for model predictive control," in *Proc. European Control Conference*, Budapest, Hungary, Aug 2009.

[26] S. Bayliss, C. S. Bouganis, and G. A. Constantinides, "An FPGA implementation of the Simplex algorithm," in *Proc. Int. IEEE Conf. on Field Programmable Technology*, Bangkok, Thailand, Dec 2006, pp. 49–55.

[27] M. S. Lau, S. P. Yue, K.-V. Ling, and J. M. Maciejowski, "A comparison of interior point and active set methods for FPGA implementation of model predictive control," in *Proc. European Control Conference*, Budapest, Hungary, Aug 2009, pp. 156–160.

[28] A. Bemporad, M. Morari, V. Dua, and E. N. Pistikopoulos, "The explicit linear quadratic regulator for constrained systems," *Automatica*, vol. 38, no. 1, pp. 3–20, Jan 2002.

[29] T. A. Johansen, W. Jackson, and P. T. Robert Schreiber, "Hardware synthesis of explicit model predictive controllers," *IEEE Trans. Control Syst. Technol.*, vol. 15, no. 1, pp. 191–197, Jan 2007.

[30] B. Fisher, *Polynomial based iteration methods for symmetric linear systems*. Baltimore, MD, USA: Wiley, 1996.

[31] (2010) Core Generator guide. Xilinx. [Online]. Available: http://www.xilinx.com/itp/xilinx6/books/docs/cgn/cgn.pdf

[32] (2010) Virtex-6 family overview. Xilinx. [Online]. Available: http://www.xilinx.com/support/documentation/data_sheets/ds150.pdf

[33] D. Boland and G. A. Constantinides, "Optimising memory bandwidth use for matrix-vector multiplication in iterative methods," in *Proc. Int. Symp. on Applied Reconfigurable Computing*, Bangkok, Thailand, Mar 2010, pp. 169–181.

[34] A. G. Richards, K.-V. Ling, , and J. M. Maciejowski, "Robust multiplexed model predictive control," in *Proc. European Control Conference*, Kos, Greece, Jul 2007, pp. 441–446.

[35] K.-V. Ling, W. K. Ho, B. F. Wu, A. Lo, and H. Yan, "Multiplexed mpc for multi-zone thermal processing in semiconductor manufacturing," *IEEE Trans. Control Syst. Technol.*, (Accepted for publication).

[36] K.-V. Ling, J. M. Maciejowski, A. G. Richards, and B. F. Wu, "Multiplexed model predictive control," *Automatica*, 2010, (Submitted).

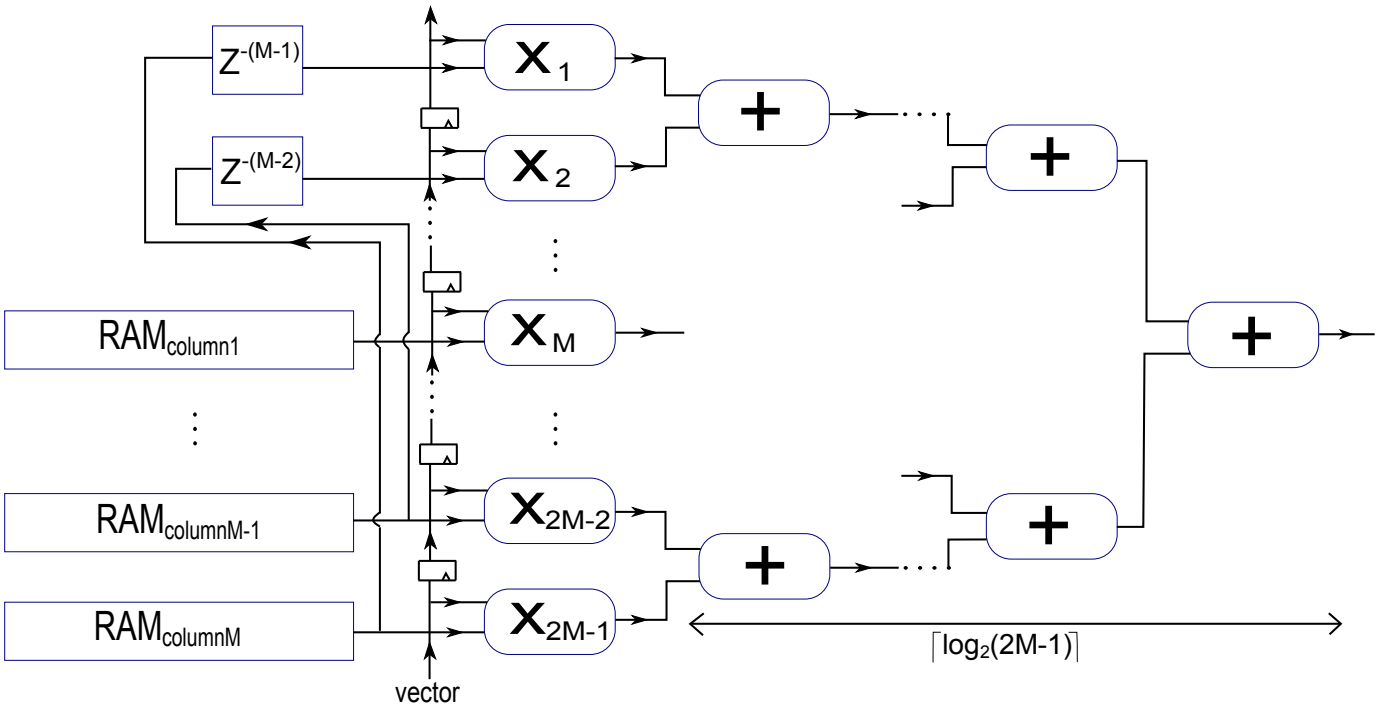


Fig. 1: Hardware architecture for computing dot-products. It consists of an array of $2M-1$ parallel multipliers followed by an adder reduction tree of depth $\lceil \log_2(2M-1) \rceil$. The rest of the operations in a MINRES iteration use dedicated components. Independent memories are used to hold columns of the stored matrix $\mathcal{A}_k$ (refer to Section VI-F for more details). z^{-M} describes a delay of M cycles.

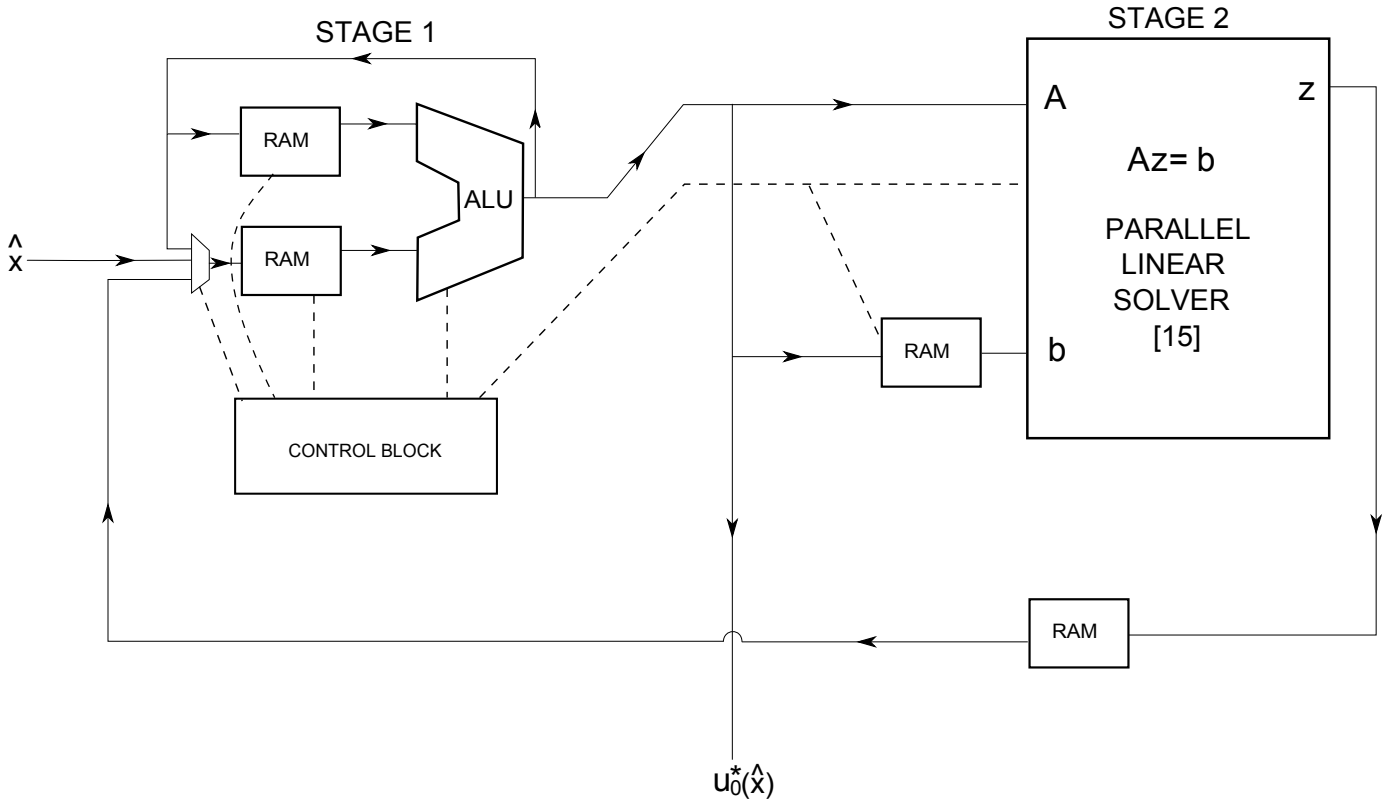


Fig. 2: Proposed two-stage hardware architecture. Solid lines represent data flow and dashed lines represent control signals. Stage 1 performs all computations apart from solving the linear system. The input is the current state measurement $\widehat{x}$ and the output is the optimal control move $u_0^*(\widehat{x})$.

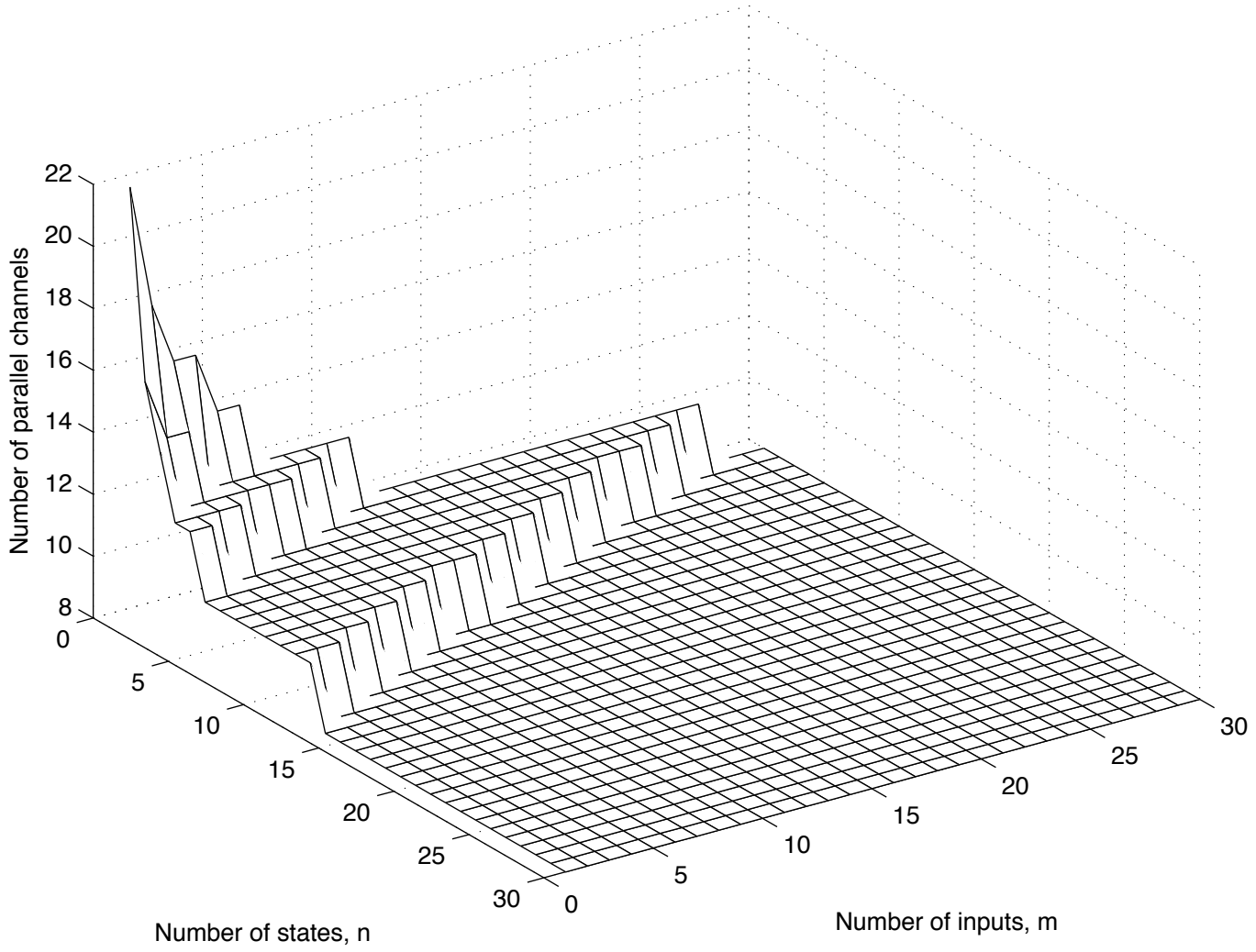


Fig. 3: Number of parallel computational channels available for problems with $T = 10$ and different number of inputs and states.

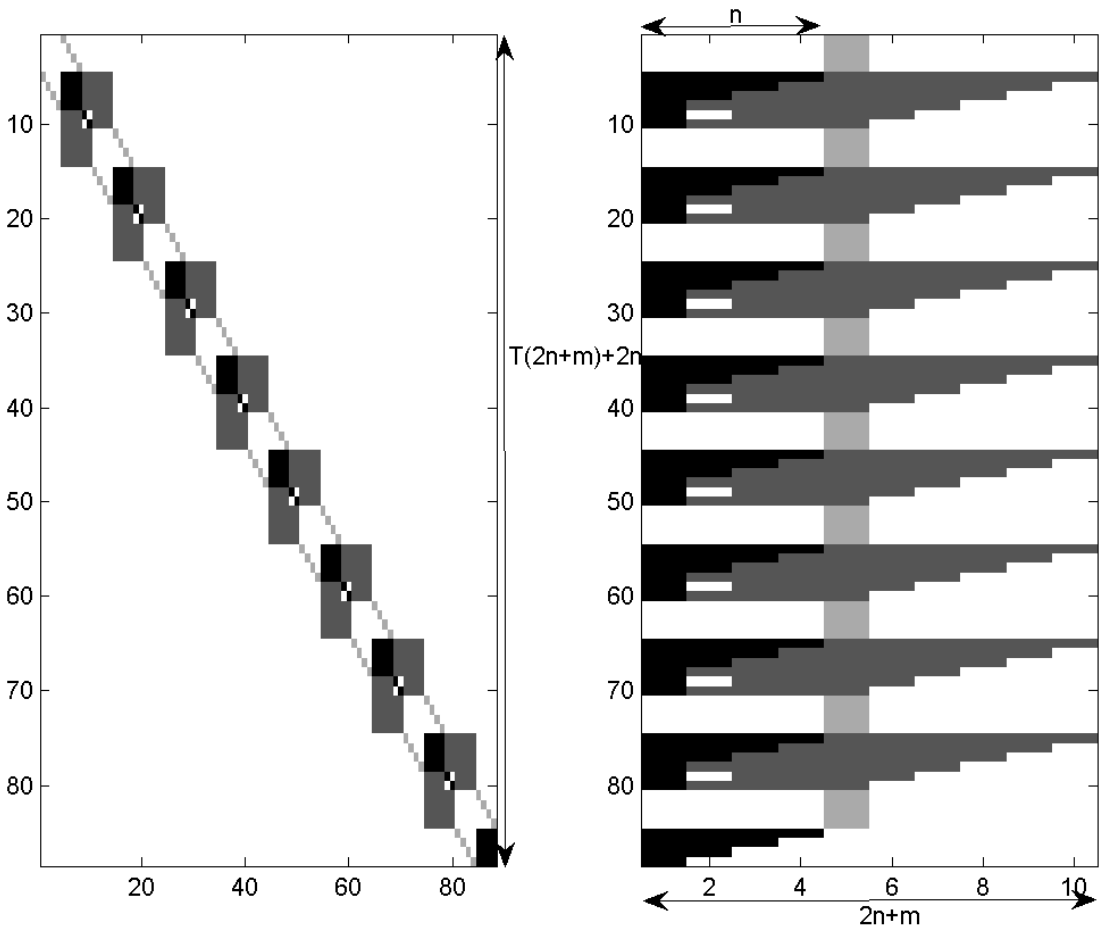


Fig. 4: Structure of original $N \times N$ matrix and corresponding $N \times M$ half-CDS matrix showing variables (black), constants (dark grey), zeros (white) and ones (light grey) for $m = 2, n = 4,$ and $T = 8$.

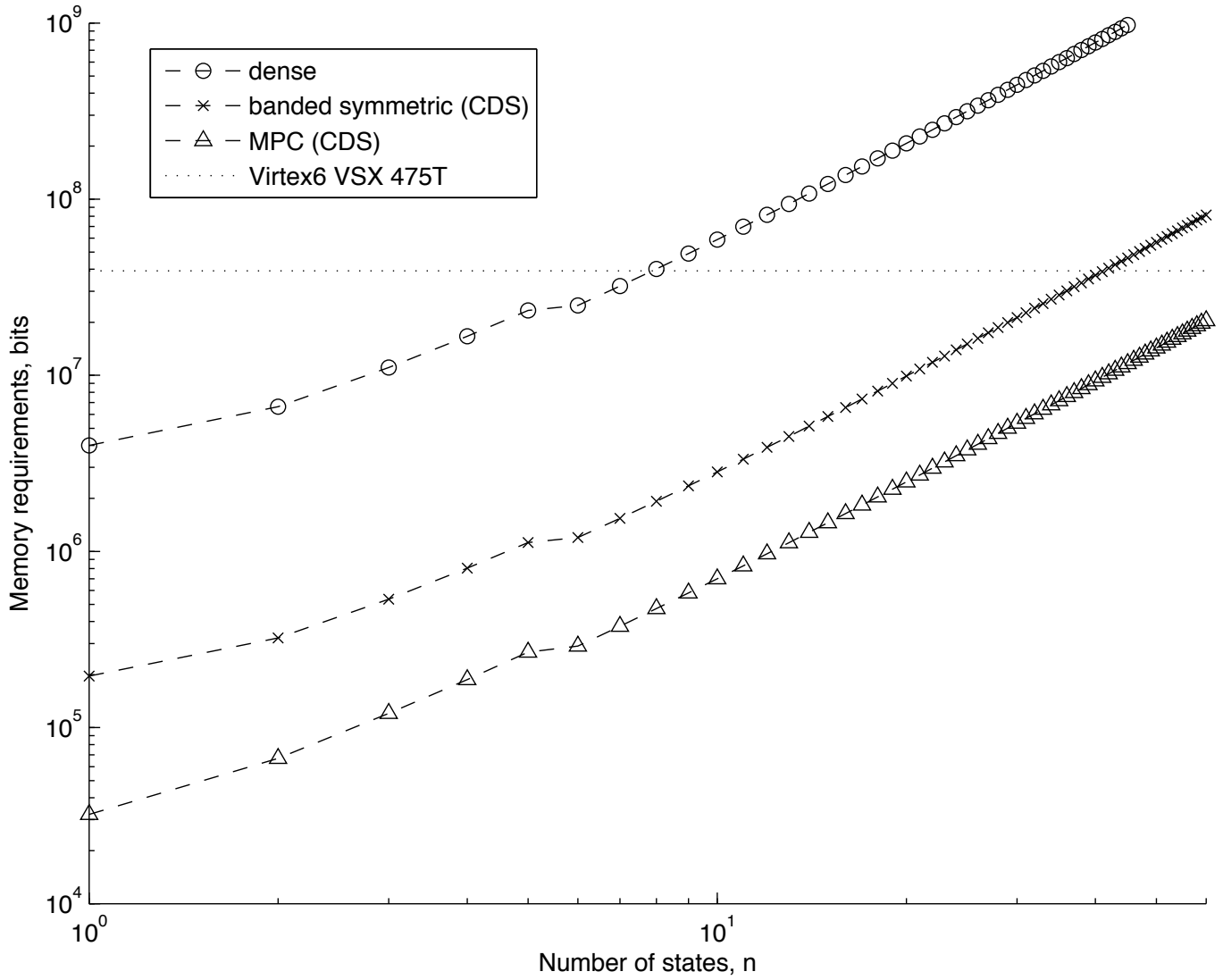


Fig. 5: Memory requirements for storing the coefficient matrices under different schemes. Problem parameters are $m = 3$ and $T = 20$. p and l do not affect the memory requirements of $\mathcal{A}_k$. The horizontal line represents the memory available in a memory-dense Virtex 6 device [32].

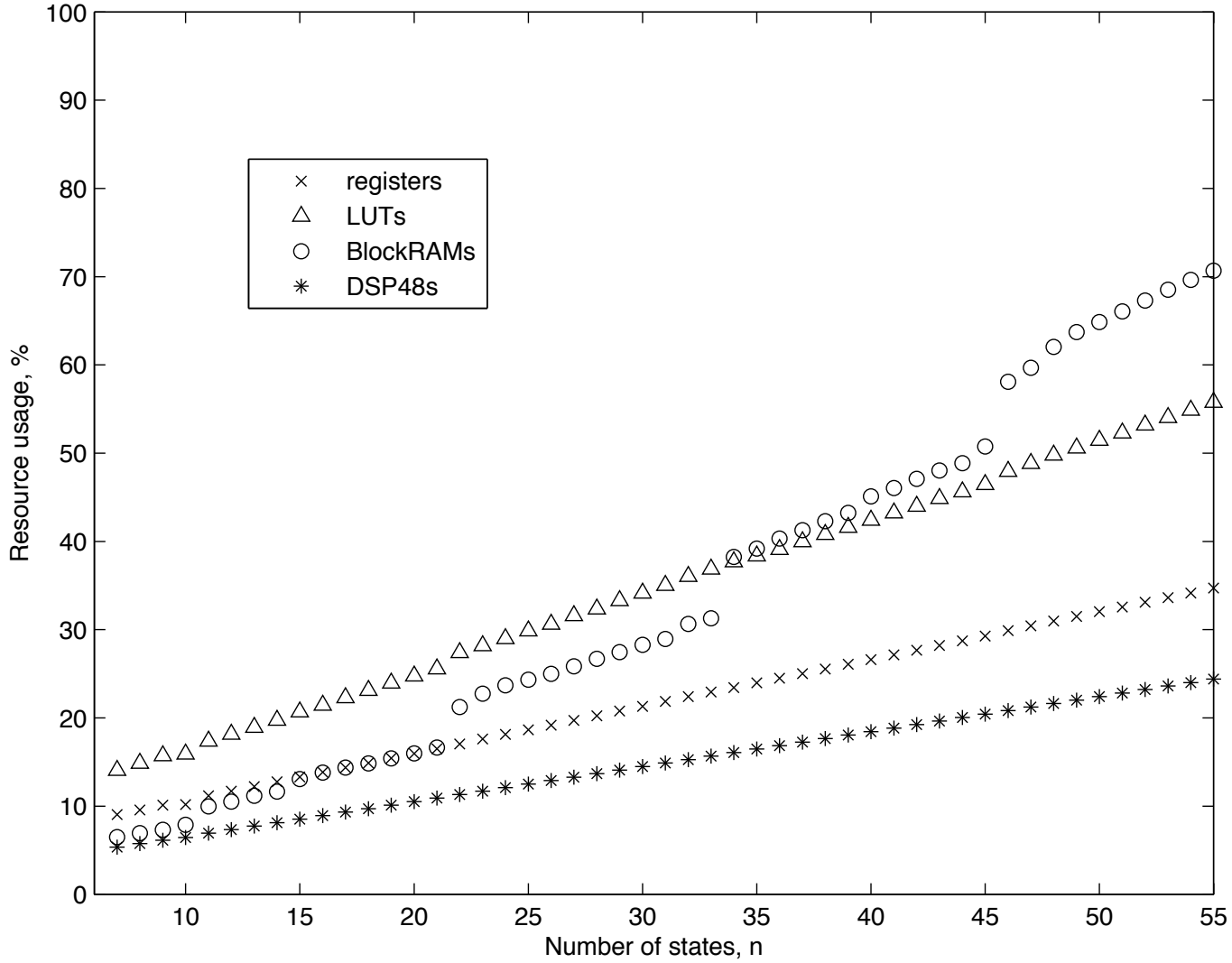


Fig. 6: Resource utilization on a Virtex 6 VSX 475T ($m = 3, T = 20, P$ given by (15)).

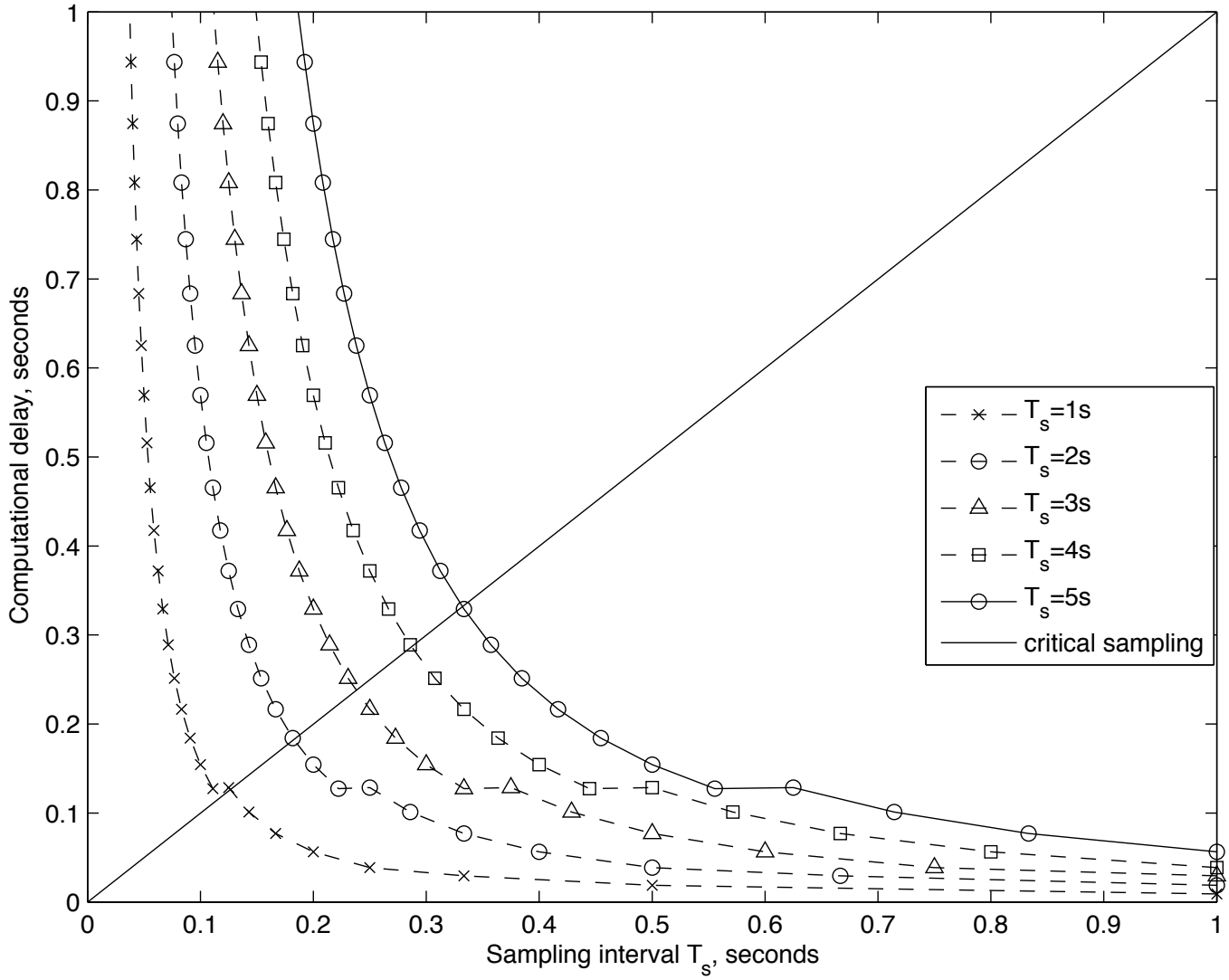


Fig. 7: Relationship between computational time and sampling interval for different horizon lengths. Problem parameters are $m = 5$, $n = 15$, $I_{MR} = N$, $I_{NW} = 20$ and $\text{FPGA}_{freq} = 150$MHz. The non-monoticity is caused by the ceil function in the expression for P.

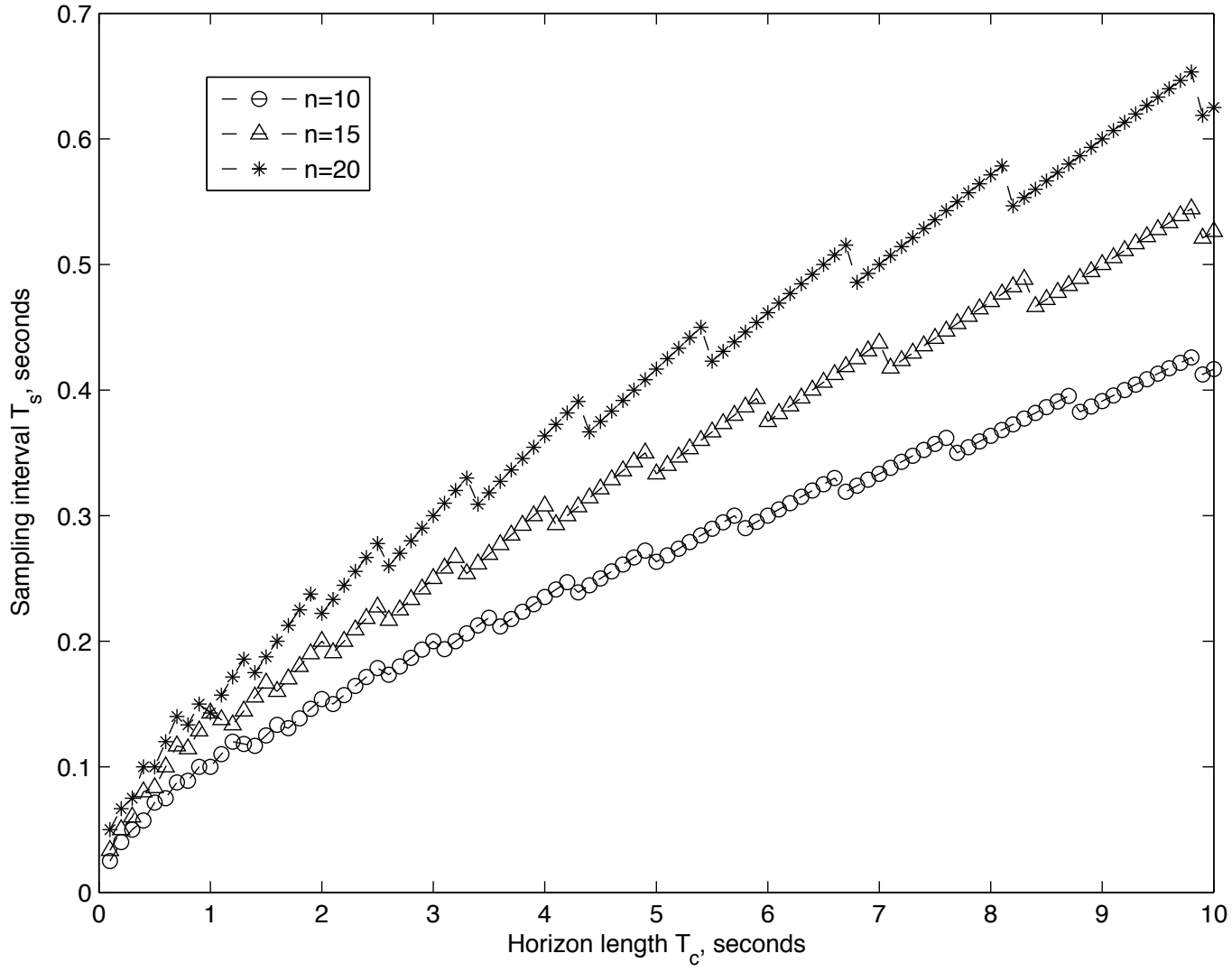


Fig. 8: Effective sampling interval against horizon length for systems with $m = 5$, $I_{MR} = N$, $I_{NW} = 20$ and different number of states.

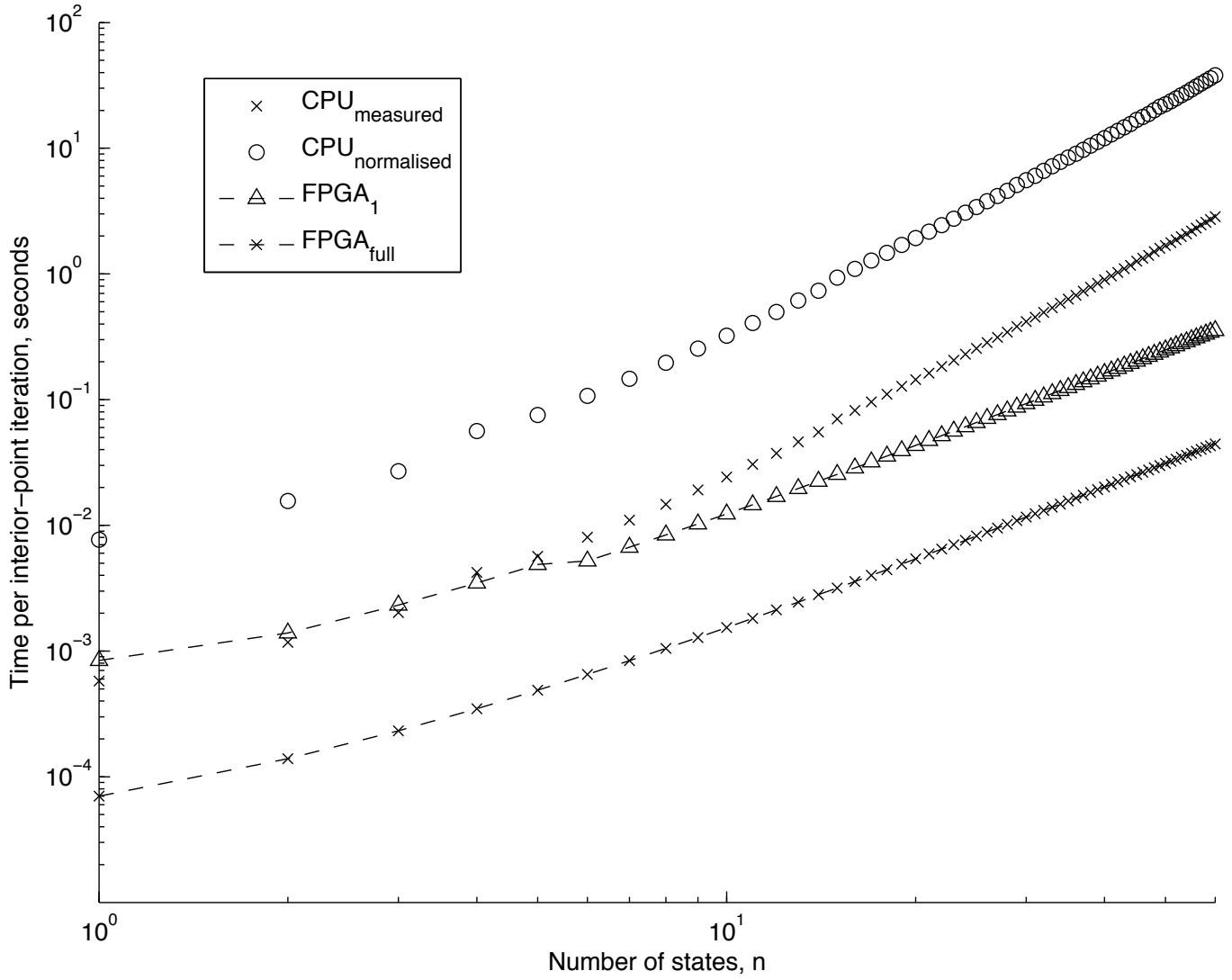
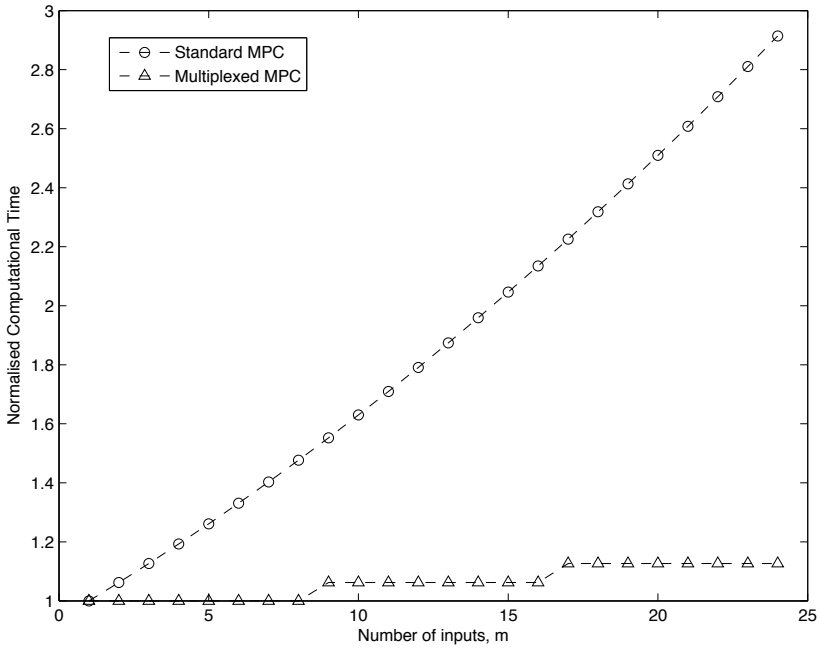
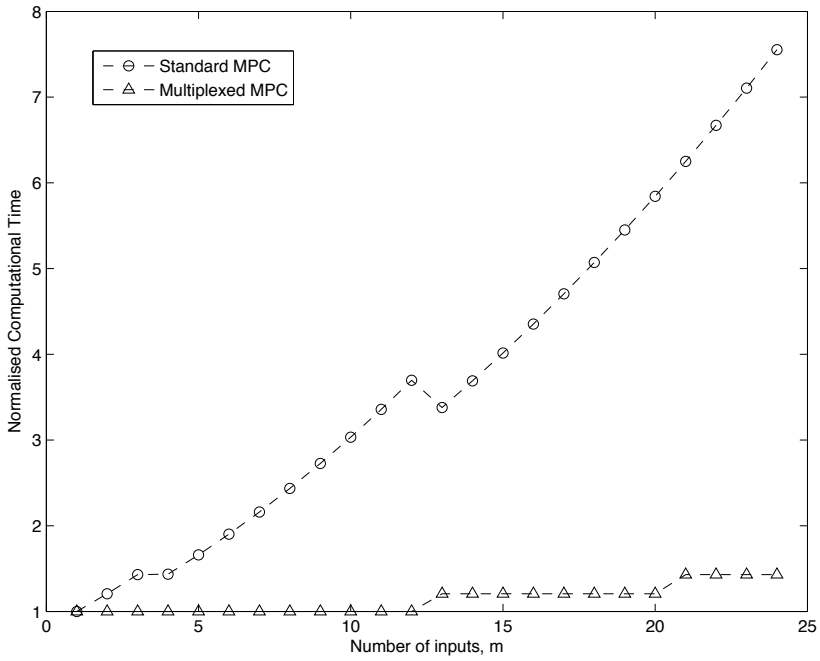


Fig. 9: Performance comparison showing measured performance of the CPU, normalized CPU performance with respect to clock frequency, and FPGA performance when solving one problem and $2P$ problems given by (15). Problem parameters are $m = 3$, $T = 20$, and $\text{FPGA}_{freq} = 150$MHz.



(a) $n = 15, T = 20$



(b) $n = 4, T = 7$

Fig. 10: Computational time reduction when employing multiplexed MPC on different plants. a) 8 parallel channels for all m, b) 14 parallel channels for $m \in (1, 3]$, 12 for $m \in (4, 12]$ and 10 for $m \in (13, 24]$. For multiplexed MPC the time required to implement the switching decision process was ignored, however, this would be negligible compared to the time taken to solve the QP problem.