

# A Custom-made Algorithmic-specific Processor for Model Predictive Control

Panagiotis Vouzis

Comp. Eng. Dept.

Lehigh University

Bethlehem, PA 18015, USA

vouzis@lehigh.edu

Leonidas G. Bleris

Electr. and Comp. Eng. Dept.

Lehigh University

Bethlehem, PA 18015, USA

bleris@lehigh.edu

Mark Arnold

Comp. Sc. Dept.

Lehigh University

Bethlehem, PA 18015, USA

maab@lehigh.edu

Mayuresh Kothare

Chemical Eng. Dept.

Lehigh University

Bethlehem, PA 18015, USA

mvk2@lehigh.edu

**Abstract**— This paper presents an algorithmic-specific processor for embedded Model Predictive Control (MPC). The optimizations associated with MPC are dominated by operations on real matrices. After analyzing the computational cost of MPC we propose connecting a limited resource host processor with an algorithmic-specific matrix processor, whose architecture is described. The matrix processor uses a 16-bit Logarithmic Number System (LNS) arithmetic unit to carry out the required arithmetic operations. The proposed architecture is implemented using a Hardware Description Language (HDL) and then synthesized and emulated on a Field Programmable Gate Array (FPGA). The timing and area cost results are presented and analyzed.

## I. INTRODUCTION

Model Predictive Control (MPC) [1] is an established control theory that is being used mainly in the chemical process industry. Due to its ability to handle Multiple-Input-Multiple-Output (MIMO) systems and to take into account constraints and disturbances explicitly, there is an increased research effort for its introduction in a wide range of nonindustrial applications. Additionally, the explicit handling of constraints by MPC makes it the control algorithm of choice for safety critical applications, such as drug delivery, automotive and aerospace control systems. As an example, in [2] the use of MPC is demonstrated for the regulation of the blood glucose concentration of a diabetic by injecting insulin according to dynamic measurements of glucose concentration.

The continuously increasing rate of integration capacity in the semiconductor industry today allows that both the controller and the system under control can be hosted on the same substrate. These Systems on a Chip (SoC) result on reduced-size devices that can exhibit low-power characteristics and can be mass produced more easily than having to assemble discrete components after fabrication. The desired characteristics of an MPC controller for such systems is to occupy small area, to exhibit low-power consumption and to be efficient enough to handle the dynamics of a system in real time. For example the recent advances in microchemical reactor fabrication, where the same substrate encompasses a chemical system with a number of actuators and sensors, pose new challenges in real-time control [3]. The efficiency of MPC for such systems is presented in [4], where the control problems of temperature distribution across a wafer and the non-isothermal flow in a microdevice are addressed.

Due to the above reasons, there has been an increased interest towards the implementation of MPC algorithms on a chip, by research institutions. A real-time implementation using an off-the-shelf processor is demonstrated in [5], where the Motorola 32-bit MPC 555 core containing a 64-bit Floating Point (FP) unit is utilized. An alternative approach for determining the optimal control moves for a system is proposed in [6], where the control law is piece-wise affine and continuous. The optimization problem is solved off-line, and during run-time the solutions are invoked from the memory. Although, this technique is proven to be very efficient in terms of performance, the memory growth is superexponential with respect to the controlled variables. Thus rendering the system prohibitive for embedded applications, where the data transfers to and from the memory is the dominant factor of power consumption. In [7], an FPGA implementation of MPC using a Quadratic Programming (QP) optimization algorithm is presented. For fast prototyping, Handel-C is used to describe the optimization algorithm in order to convert it to a hardware description format, which is simulated in conjunction with Matlab, before being downloaded onto an FPGA.

Alternatively, we propose a custom hardware architecture consisting of a general purpose microprocessor and an auxiliary unit, tailored to accelerate computationally demanding MPC operations. The general purpose microprocessor acts as the master in the system, i.e. it carries out the tasks of Input/Output (I/O), initializes and sends the appropriate commands to the auxiliary unit and receives back the optimal control moves. The auxiliary unit acts as a matrix coprocessor by carrying out matrix operations, such as addition, multiplication, inversion etc. This algorithmic-specific stores locally the intermediate results and the matrices involved in the MPC algorithm, and it communicates with the microprocessor only for initialization and for sending back the results of the MPC algorithm, thus the communication overhead is minimized between the two units. Additionally, the Logarithmic Number System (LNS) [8] is used by the coprocessor to carry out the arithmetic operations required. The utilization of LNS allows the reduction of the required word-length to 16 bits, and consequently a general purpose microprocessor of the same word-length is used. The alternative of an equivalent Floating Point (FP) unit, although it exhibits similar delay to the LNS,

is proven to occupy 40% more area [9], and thus consumes more power.

The design path for the proposed architecture follows a co-design methodology, which is an intermediate approach for implementing algorithms between hardware (H/W) and software (S/W). Co-design combines S/W's flexibility with the high performance offered by H/W, by implementing the computationally intensive parts in H/W, while using S/W to carry out algorithmic control tasks and high level operations. The computationally demanding parts of MPC, determined by performing a profiling study of the algorithm, are migrated to the matrix coprocessor described above, while the rest of the algorithm is hosted by the general purpose microprocessor. The whole design is described by means of a Hardware Description Language (HDL), and after synthesis a Field Programmable Gate Array (FPGA) is used to verify the functionality of the controller. During the development process the Hardware-In-the-Loop (HIL) technique is used in order to test and debug the design.

## II. MODEL PREDICTIVE CONTROL

MPC is an algorithm that uses a model describing the system under control. Initially, at time step  $t$ , the model is used to predict a series of  $k$  future outputs of the system up to time  $t+k$ , i.e.  $y(t+k|t)$  for  $k = 1, \dots, P$ . The next step is to calculate  $M$  optimal future signal,  $u(t+k|t)$  for  $k = 1 \dots M$ , in order the process to follow a desired trajectory  $y_{\text{ref}}$  as closely as possible. The parameters  $P$  and  $M$  are referred to as prediction and control horizon, and their values affect the quality performance of the controller.

The criterion for the optimal future moves is usually a quadratic cost function of the difference between the predicted output signal and the desired trajectory, which can include the control moves  $u(t+k|t)$  in order to minimize the control effort. A typical objective function has the form:

$$J_P(k) = \sum_{k=0}^P \{ [y(t+k|t) - y_{\text{ref}}]^2 + Ru(t+k|t)^2 \} \quad (1)$$

$$|u(t+k|t)| \leq \mathbf{b}, \quad k \geq 0 \quad (2)$$

where  $R$  is a design parameter used to weight the control moves, and  $\mathbf{b}$  is the constraint vector that the future inputs have to obey. Out of the  $M$  moves given by the minimization of the objective function, only the first one is used; the rest are discarded since at the next sampling instant the output is measured and the procedure is repeated with the new measured values.

The future optimal moves are based on the minimization of the objective function (1) which can be achieved with different optimization algorithms. In this paper we use the Newton's optimization method based on a state-space model of the system given by:

$$\mathbf{x}(t+1) = \mathbf{A}\mathbf{x}(t) + \mathbf{B}\mathbf{u}(t) \quad (3)$$

$$\mathbf{y}(t) = \mathbf{C}\mathbf{x}(t), \quad (4)$$

where  $\mathbf{x}$  is the state and  $\mathbf{A}, \mathbf{B}, \mathbf{C}$  are the matrices describing the model of the system, resulting in the prediction model

$$\hat{y}(t+k|t) = C\hat{x}(t+k|t) \quad (5)$$

$$\hat{y}(t+k|t) = C[\mathbf{A}^k x(t) + \sum_{i=1}^k \mathbf{A}^{t-1} \mathbf{B} u(t+k-i|t)]. \quad (6)$$

## III. COMPUTATIONAL ANALYSIS OF MPC

The problem of (1) can be solved numerically approximating  $J_P(k)$  by a quadratic function around  $u$ , obtaining the gradient  $\nabla(J_P)$  and the Hessian  $\mathbf{H}(J_P)$ , and iterating

$$u(t+1) = u(t) - \mathbf{H}(J_P)^{-1} \cdot \nabla(J_P) \quad (7)$$

where

$$\nabla(J_P) = 2(\mathbf{B}^T \mathbf{B} + R\mathbf{I})\mathbf{u} + 2\mathbf{B}^T \mathbf{A}\mathbf{x} - 2\mathbf{B}^T \mathbf{y}_{\text{ref}} + \mu \mathbf{I}_M \Phi \quad (8)$$

$$\mathbf{H}(J_P) = 2(\mathbf{B}^T \mathbf{B} + R\mathbf{I}) + 2\mu \mathbf{I}_M \Psi \quad (9)$$

$$\Phi = \left[ \frac{1}{(u_1 - b)^2} - \frac{1}{(u_1 + b)^2} \cdots \frac{1}{(u_M - b)^2} - \frac{1}{(u_M + b)^2} \right]^T \quad (10)$$

$$\Psi = \left[ \frac{1}{(u_1 - b)^3} + \frac{1}{(u_1 + b)^3} \cdots \frac{1}{(u_M - b)^3} + \frac{1}{(u_M + b)^3} \right]^T \quad (11)$$

$\mathbf{I}$  is an  $M \times M$  identity matrix,  $\mathbf{I}_M = [11 \dots 11]$  is a vector of length  $\mathbf{M}$ , and  $\mu$  is a parameter that is adjusted appropriately in order the method to converge.

It is apparent that Newton's method is characterized by abundant matrix operations like matrix by vector multiplications, matrix additions and a matrix inversion whose computational complexity depends on the the size of the control horizon  $M$ , and on the number of states  $N$ .

### A. The Co-design Path

The previous analysis of the operations required for each optimization step of MPC reveals the reason why only high-performance processors with increased precision are used to solve such a problem. A small-precision system, may fail due to accumulated errors of the arithmetic operations involved, while strict real-time constraints are difficult to be met with today's microprocessors for embedded applications. However, a custom designed architecture tailored to this particular algorithm can address these problems, while consuming low power and achieving sufficient throughput for real-time applications.

Towards this direction, a co-design methodology is used to develop an architecture that proves to be efficient both in power consumption and performance, while it is sufficiently flexible to be embedded in bigger systems that need to include MPC in their functionality. In Fig. 1 the design path adopted in this work is presented. Initially, the specifications are set and the software-hardware partitioning follows. The decision upon the partitioning is based on a profiling study of MPC which helps to identify the computationally demanding parts of the algorithm. After the communication protocol between the two parts is specified, they are implemented using a Hardware Description Algorithm (HDL) for the H/W and

a high-level programming language for the S/W. The next step is to co-simulate the two parts in order to verify the correct functionality and the performance of the complete design. If the verification process fails then a backward jump is made to the appropriate design step, e.g. if the performance of the system does not meet the specifications, then a new partitioning decision is made and the whole design path is repeated.

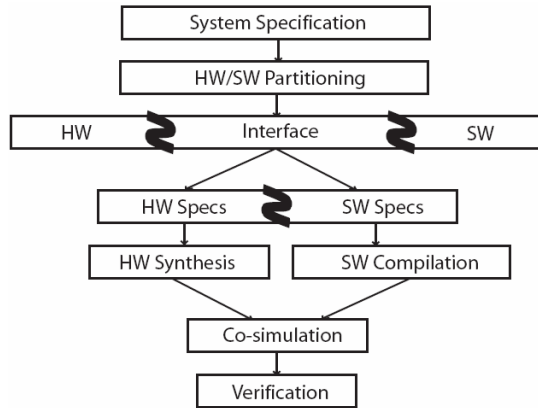


Fig. 1. The codesign path.

### B. The Profiling Study

The most critical step in the co-design path is the H/W-S/W partitioning, because this decision determines both the amount of speed up achieved by the H/W, and how flexible the whole system is. This decision relies on a profiling study of the MPC optimization algorithm described by Eq. (8)–(11).

The system used as a benchmark it taken from [5], and it is a rotating antenna driven by an electric motor. It has the following state-space form:

$$x(k+1) = \begin{bmatrix} 1 & 0.1 \\ 0 & 0.9 \end{bmatrix} x(k) + \begin{bmatrix} 0 \\ 0.1k \end{bmatrix} u(k) \quad (12)$$

$$y(k) = [1 \ 0]x(k) \quad (13)$$

Initially, the percentage of time spent on each of these equations is determined. Fig. 2 depicts this distribution of time for a fixed prediction horizon of  $P = 20$  and variable control horizon  $M$ . It can be observed that apart for the case  $M = 2$ , the Gauss-Jordan inversion algorithm is the main bottleneck of the problem. Additionally, for  $M \geq 8$ , the calculation of the Hessian becomes increasingly more time consuming compared to the Gradient function. Since the computational complexity of the Hessian consists almost entirely of the  $\Psi$  (the factor  $2(\mathbf{B}^T\mathbf{B} + \mathbf{R}\mathbf{I})$  is precomputed and only invoked from the memory) we can conclude that the cubings and inversions add a substantial burden to the computational effort. The initialization of the algorithm takes place once, at the beginning of the simulation, thus its contribution to the total computational cost is unobservable. Although, this analysis is performed on a Pentium processor, which incorporates

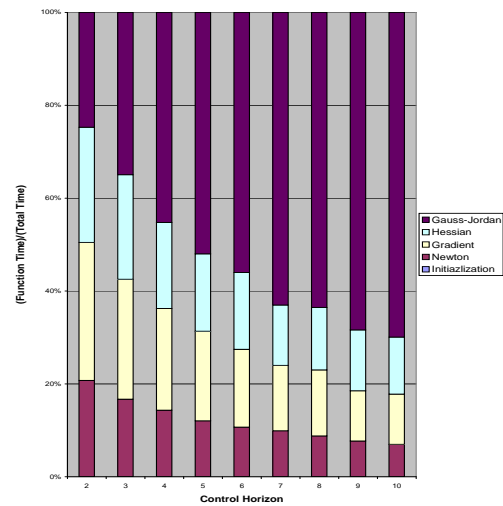


Fig. 2. Profiling results for the antenna control problem on a Pentium processor.

a double precision FP unit, it is safe to assume that are representative of any implementation that uses a FP unit.

The critical design choice is to determine an architecture that performs well both in operations involved in matrix manipulations, such as multiply-accumulate, and in real-number squarings, cubings and inversions, included in the evaluation of  $\Phi$  and  $\Psi$ . The standardized arithmetic system for real-number arithmetic is the FP. Nevertheless, the profiling results urge us to consider other options that may fit better to the particular nature of the problem. The Logarithmic Number System (LNS) has appeared as an alternative to FP arithmetic [8], and recent studies prove that for word lengths up to 32-bits, LNS is more efficient than FP, with an increasing efficiency as the word length decreases [9]. More precisely, for a word length of 16-bits, which is sufficient for a number of MPC problems, LNS occupies 40% less area compared to FP, while the delay between the two arithmetic systems is similar. Additionally, in LNS the calculation of a real-number's square, cube and inverse can be implemented very efficiently, since they are reduced down to the simple operations of left-shifts additions and two's complements, with simpler faster and less power hungry circuits than the ones in an equivalent FP unit.

## IV. THE LOGARITHMIC NUMBER SYSTEM

In LNS, a real number,  $X$ , is represented by the logarithm of its absolute value,  $x$ , and an additional bit,  $s$ , denoting the sign of the number  $X$ :

$$x = \{s, \text{round}(\log_b(|X|))\}, \quad (14)$$

where  $s = 0$  for  $X > 0$ ,  $s = 1$  for  $X < 0$ , and  $b$  is the base of the logarithm. The  $\text{round}()$  operation approximates  $x$  so that it is representable by  $N = K + F$  bits in two's complement format. The number of integer  $K$  and fractional  $F$  bits is a design choice that determines the dynamic range and the accuracy respectively. In this work  $K = 6$  and  $F = 9$  is used.

In LNS real-number multiplication, division, squaring, cubing and inverse are simplified considerably compared to a fixed-point representation, since they are converted to addition, subtraction, shifting, shifting and addition and negation respectively. The operations of addition and subtraction, though, are more expensive, and they account for most of the delay and area cost of an LNS implementation. A simple algorithm usually used is described by:

$$\log_b(X + Y) = \log_b\left(X\left(1 + \frac{Y}{X}\right)\right) = x + s_b(z) \quad (15)$$

$$\log_b(X - Y) = \log_b\left(X\left(1 - \frac{Y}{X}\right)\right) = x + d_b(z). \quad (16)$$

where  $z = |x - y|$ ,  $s_b = \log_b(1 + b^z)$  and  $d_b = \log_b(1 - b^z)$ . A naïve implementation for these functions is to store them in Look-Up Tables (LUTs) with respect to all the possible values of  $z$ . However, there are different optimization techniques that reduce considerably the required memory size. The one adopted in this work is the technique of co-transformation [10] where two additional functions are stored, but overall considerable memory savings are achieved, without compromising any accuracy in the final result.

## V. THE MICROPROCESSOR-COPROCESSOR ARCHITECTURE

In order to make the design flexible and easily embeddable an off-the-self general purpose microprocessor is used. This choice facilitates the development, since a high-level programming language can be used for its programming, and the peripherals that are usually available, such as serial communication (RS-232), Digital-to-Analog (D/A) conversion, render the microprocessor the communication interface between the matrix coprocessor, that calculates the optimal moves, and the plant under control.

### A. Communication

The matrix coprocessor acts as a peripheral device of the microprocessor by having dedicated part of the address space, which is limited to two memory locations since the microprocessor needs to send commands and data, and read back the available data and the status of the coprocessor. Additionally, four more signals are used: Chip-Select (CS) to signal the coprocessor's selection, Read (RD) to signal reading, Write (WR) to signal writing, and Data-of-Status (DS) to distinguish between data and status. The data exchanged between the two ends is divided in two parts. The first part consists of the required matrices used in every optimization step. These matrices are sent only at the beginning of the algorithm and are stored locally in the coprocessor, thus this communication overhead is negligible. The second part includes the sequence of commands that comprise the optimization algorithm, and the optimal move that is sent back to the microprocessor at the end of each optimization step.

The matrix coprocessor works independently of the microprocessor, hence during the execution of a command the microprocessor can perform any other task and send the next command or read back available data whenever is desirable.

This configuration accelerates considerably the execution of MPC, since otherwise a microprocessor is dedicated to perform all the computationally demanding matrix operations described by Eq. (8)–(11).

### B. The Datapath of the Matrix Coprocessor

The main burden of performing the computations required for MPC, falls on the coprocessor. This unit has to demonstrate enough performance to meet the requirements of a tight real-time application, and at the same time it has to consume low-power in order to be suitable for embedded applications. Additionally, it is desirable to be scalable in order to address problems of a wide range of sizes without wasting resources. The systolic architectures for matrix operations, such as multiplication or inversion, appear to be very efficient in terms of performance. In [11] a systolic architecture for real matrix inversion or size  $n$  is proposed, requiring  $2n^2 - n$  processing elements implementing the operations of  $c - ab$ ,  $ab - c$  or  $1/c$ . Due to the highly parallel nature of this architecture the time units required for a matrix inversion are  $(5n - 1)t_d$ , where  $t_d$  is the delay of a real scalar inversion. However, it is not scalable to problems of different size, thus wasting resources when matrices smaller than  $n$  are manipulated.

Instead, we propose a coprocessor that manipulates matrices in an iterative fashion, i.e. there is one LNS arithmetic unit, that can execute the operation of  $ab + c$ , and a number of local memory units that store intermediate results. The sequence of operations are controlled by a Finite State Machine (FSM) that receives a command from the microprocessor and after executing the necessary tasks signals back the completion.

The processors datapath includes two matrix registers, **A** and **C**, each of which contains an  $n \times n$  matrix, where  $n \leq 2^m$  and the constant  $m$  determines the maximum-sized matrix processed by a single command. The host sends  $n$  to the matrix processor, which then defines the size of the matrices needed to execute a particular operation of the MPC model. **A** and **C** both are  $16 \times 2^{2m}$  memories. **A** is a dual-port memory, where one of its ports is attached to a pipelined LNS ALU of the kind described in [10]. In the current prototype, there is only one such ALU, but the highly independent nature of common matrix computation may allow up to  $2^m$  such ALUs to be used effectively in future versions. When  $n < 2^m$ , there will be unused rows and columns in **A** and **C** which the processor ignores. The address calculation for  $\mathbf{A}_{i,j}$  or  $\mathbf{C}_{k,j}$  simply involves concatenation of the row and column indices. Such indices are valid in the range  $0 \leq i, j, k < n$ .

In addition to the matrix registers, **A** and **C**, the processor has a main memory,  $m[ ]$ , to store the several matrices needed by MPC. When the host requests the matrix processor to perform a command, say matrix multiply, the host also sends `addr`, which indicates the base address of the other operand in memory, referred to as **B**. Unlike **A** and **C**, which have unused rows and columns, **B** is stored with conventional row-major order; an  $n \times n$  matrix takes  $n^2$  rather than  $2^{2m}$  words. The preferred mode of operation is to keep all the matrices required inside the matrix processor. In the event  $m[ ]$  is

TABLE I

THE COMMAND SET OF THE MATRIX COPROCESSOR.

| Mnem.    | $O(1)$ Oper.   | Mnem.   | $O(n^2)$ Oper.                                   |
|----------|--|---------|--|
| PUTN     | $n \leftarrow \text{host}$                           | INPC    | $\mathbf{C} \leftarrow \text{host}$              |
| PUTRC    | $\{r, c\} \leftarrow \text{host}$                    | OUTC    | $\text{host} \leftarrow \mathbf{C}$              |
| PUTA     | $\mathbf{A}_{i,j} \leftarrow \text{host}$            | OUTA    | $\text{host} \leftarrow \mathbf{A}$              |
| GETA     | $\text{host} \leftarrow \mathbf{A}_{i,j}$            | LOADC   | $\mathbf{C} \leftarrow \mathbf{B}$               |
| PIVOT    | $x \leftarrow -1/\mathbf{A}_{i,i}$                   | LOADA   | $\mathbf{A} \leftarrow \mathbf{B}$               |
|          | $\mathbf{A}_{i,i} \leftarrow 1$                      | STOREC  | $\mathbf{B} \leftarrow \mathbf{C}$               |
| Mnem.    | $O(n)$ Oper.   | STOREAZ | $\mathbf{B} \leftarrow \mathbf{A}$               |
| LOADVA   | $\mathbf{A}_i \leftarrow \mathbf{b}$                 |         | $\mathbf{A} \leftarrow \mathbf{0}$               |
| STOREVA  | $\mathbf{b} \leftarrow \mathbf{A}_i$                 | STOREAI | $\mathbf{B} \leftarrow \mathbf{A}$               |
| STOREVAZ | $\mathbf{b} \leftarrow \mathbf{A}_i$                 |         | $\mathbf{A} \leftarrow \mathbf{I}$               |
|          | $\mathbf{A}_i \leftarrow \mathbf{0}$                 | ADDX    | $\mathbf{A} \leftarrow \mathbf{A} + x\mathbf{B}$ |
| ADDXVA   | $\mathbf{A}_i \leftarrow \mathbf{A}_i + x\mathbf{b}$ | MULX    | $\mathbf{A} \leftarrow x\mathbf{A}$              |
| SUMVA    | $x \leftarrow \sum_{i=1}^n \mathbf{A}_i$             | MULV    | $\mathbf{A} \leftarrow \mathbf{C}\mathbf{b}$     |
| POW2A    | $\mathbf{A}_i \leftarrow 1/\mathbf{b}^2$             |         |  |
| POW3A    | $\mathbf{A}_i \leftarrow 1/\mathbf{b}^3$             |         |  |

inadequate for a particularly large MPC problem, commands are provided to transfer  $n^2$  words at the maximum achievable rate by the host interface.

Table I shows the operations supported by the matrix processor. “host” indicates a data transfer to/from the microprocessor,  $i$  and  $j$  are indices provided by the host,  $\mathbf{A}_i$  is a row chosen by the host,  $\mathbf{b}$  and  $\mathbf{B}$  are vectors and matrices, respectively, (stored in  $m[ ]$  at the base address specified by the host),  $\mathbf{I}$  is the identity matrix,  $\mathbf{0}$  is the zero matrix, and  $x$  is an LNS scalar value.

The matrix processor consists of one-hot controller and associated datapath generated by a tool called VITO [12]. One-hot encoding, in which there is one register to each state in the finite-state machine, is suitable when combinational logic is more expensive than registers, which is the case of FPGA devices. The advantage of VITO is that it allows us to develop a complex state machine, such as our matrix processor, with almost as much ease as if we were developing software [13].

## VI. IMPLEMENTATION RESULTS

For implementation purposes we selected the 16-bit Extensible Instruction Set Computer (EISC) of ADCUS to act as the host [14]. Both the microprocessor and the coprocessor are described in Verilog and were simulated by using ModelSim XEIII/Starter 6.0a in order to verify the functionality and measure the performance of the system. The target technology for synthesis is a Field Programmable Gate Array of Xilinx, thus the development environment ISE 7.1 of the same vendor is used. The program running on the microprocessor is developed in the C programming language by using the EISC Studio of ADCUS, and for prototyping the board ML401, hosting an XC4VLS25-FF668-10C Virtex-IV FPGA of Xilinx, is used.

In Table II the breakdown of the synthesis results is presented. The two BlockRAMs of the coprocessor are embedded memory blocks into the FPGA and are used to instantiate the matrix registers  $\mathbf{B}$  and  $\mathbf{C}$  analyzed in Section V-B, while the three BlockRAMs of the microcontroller contain the program code. In the same table the synthesis results from [7] are presented for comparison.

TABLE II

THE BREAKDOWN OF THE SYNTHESIS OF THE PROPOSED ARCHITECTURE.

| Characteristic | Coproc. | $\mu\text{C}$ | [7]    |
|----------------|---------|---------------|--------|
| Slices         | 1961    | 3273          | 3301   |
| LUTs           | 3457    | 4967          | 5916   |
| Flip-Flops     | 1444    | 1135          | 1596   |
| BlockRAMs      | 2       | 3             | 17     |
| Frequency      | 25 MHz  | 25 MHz        | 28 MHz |
| Program Mem.   | N/A     | 2.7 Kb        | N/A    |

Although, the total area of the coprocessor and the microprocessor is 58% bigger than the one in [7], for comparison purposes the area of only the coprocessor should be considered, because it is assumed that a typical embedded application encompasses a general purpose microprocessor. Thus, in order to implement efficiently an MPC algorithm with the proposed architecture the area overhead consists of only the coprocessor. It has to be noted that the comparison is done only in terms of the slices, since LUTs, Flip-Flops, BlockRAMs and wiring are contained in them. We can see that the clock frequency of the design in [7] is 12% faster than our design, but the area that occupies is 68% bigger. Since the area of a design is one of the most decisive factors of its power consumption, the architecture proposed here is much less power hungry than the one in [7], with a penalty in performance.

However, the comparison between these two works cannot be done by just considering the two factors of area and clock frequency; there are other issues that is difficult to be taken into account. For example, in [7] a QP-optimization algorithm is used, while in this work the Newton’s method is utilized. Additionally, the two works use a different accuracy for the arithmetic calculations. In [7] a 27-bit FP unit is used, but we found sufficient to use a 16-bit LNS unit for the antenna problem presented in VI-A. Moreover, the final performance of our design depends on the control horizon and on the number of optimization iterations used to solve the problem, which consequently have an impact on the controller’s output. Thus, there may be applications that one or the other design is more suitable, and this is a matter of further study.

### A. Case study: Rotating Antenna

The functionality of the system was verified both in simulation and on the FPGA. For the FPGA case the Hardware-In-the-Loop technique was used by interfacing the evaluation board, via the RS-232 protocol, with Matlab running on a workstation. At the beginning of the simulation Matlab sends the initialization matrices describing the model under simulation, the desired setpoint for the output, and the number of optimization iterations required. On the FPGA side the RS-232 protocol is implemented by the microcontroller, which forwards the data to the coprocessor and initiates the execution of Newton’s algorithm. After the optimal move is calculated, it is sent to Matlab, which responds by sending the feedback of the model’s output and the desired setpoint.

The system used for HIL simulation is the one described in Section III-B, and was simulated for different values of control

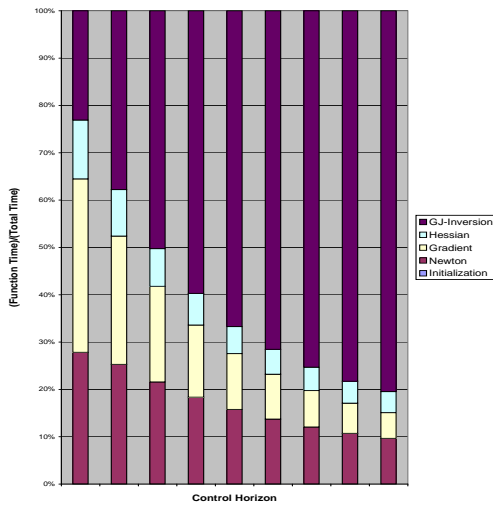


Fig. 3. Profiling results for the antenna control problem on the processor-coprocessor design.

horizon,  $M$ , while the prediction horizon is equal to  $P = 20$ . For the proposed architecture profiling studies were conducted, respective to the ones in Section III-B, and are presented in Fig. 3. It can be seen that as  $M$  increases the inversion of the Hessian matrix, which is done by the Gauss-Jordan algorithm, is again the bottleneck. Additionally, Table III presents the clock cycles required by each function of the MPC algorithm for  $M \in [2, 10]$ .

For a clock frequency of 25MHz, that we are using, the delay for one optimization iteration, for  $M = 3$ , is equal to  $D_{opt.} = 0.89ms$ . The same control problem solved by using Motorola's 32-bit MPC 555 processor, running at 40MHz and incorporating a double precision FP unit, requires 15ms for the same control horizon [5]. Thus, this proposed custom-designed architecture offers a significant improvement in the performance for implementing MPC. Moreover, the work presented in [5], which is based on a general purpose processor, uses redundant resources to solve the particular problem, i.e., the double precision FP offers unneeded accuracy for the arithmetic calculations, thus making it inefficient in terms of power consumption.

TABLE III

THE CLOCK CYCLES REQUIRED BY EACH FUNCTION OF THE NEWTON'S ALGORITHM FOR ONE OPTIMIZATION ITERATION.

| $M$ | Newton | Gradient | Hessian | GJ-Inversion | Total  |
|-----|--------|----------|---------|--------------|--------|
| 2   | 4603   | 6061     | 2054    | 3821         | 16539  |
| 3   | 5647   | 6061     | 2194    | 8445         | 22347  |
| 4   | 6510   | 6096     | 2404    | 15174        | 30184  |
| 5   | 7377   | 6131     | 2684    | 24018        | 40210  |
| 6   | 8275   | 6201     | 3003    | 35056        | 52535  |
| 7   | 9245   | 6378     | 3538    | 48187        | 67348  |
| 8   | 10178  | 6495     | 4170    | 63606        | 84449  |
| 9   | 11111  | 6612     | 4802    | 81208        | 103733 |
| 10  | 12122  | 6834     | 5572    | 101133       | 125661 |

## VII. CONCLUSIONS

In this work we present a systematic approach towards the implementation of MPC on a chip. The conducted code-sign analysis revealed that there are certain repetitive matrix manipulations that dominate the arithmetic operations of the Newton's optimization algorithm. A codesign methodology is utilized that results in an architecture consisting of a general purpose microprocessor and a matrix coprocessor. The coprocessor acts as a hardware accelerator to the microprocessor by performing all the computationally demanding operations. The arithmetic number system of choice is LNS, since, for a wordlength of 16 bits, is more efficient in terms of power consumption compared to the FP number system. The coprocessor manipulates the matrices in an iterative fashion in order to be flexible and scalable to the size of any kind of control problem. For the implementation of the FSM, consisting the control unit of the coprocessor, VITO is used which enables us to describe one-hot FSMs using software-like coding. The final design, after synthesis, has been emulated on an FPGA and a comparison, over a previous SoC implementation of MPC, demonstrates a tradeoff between area and performance. Compared to a pure software implementation on a general purpose microprocessor the proposed architecture is superior both in terms of performance and power consumption. An alternative FPGA implementation of MPC although exhibits 12% better clock frequency, it occupies 68% more area.

## REFERENCES

- [1] E. F. Camacho and C. Bordons, *Model Predictive Control*. Springer, New York, 1999.
- [2] L. G. Bleris and M. Kothare, "Implementation of Model Predictive Control for Glucose Regulation on a General Purpose Microprocessor," in *44<sup>th</sup> IEEE Conference on Decision and Control and European Control Conference, ECC 2005*, (Seville, Spain), Dec. 12–15 2005.
- [3] K. F. Jensen, "Microchemical Systems: Status, Challenges, and Opportunities," *AIChE Journal*, vol. 45, pp. 2051–2054, Oct. 1999.
- [4] L. G. Bleris, J. G. Garcia, M. V. Kothare, and M. G. Arnold, "Towards Embedded Model Predictive Control for System-on-a-Chip Applications," *Journal of Process Control*, vol. 16, pp. 255–264, March 2006.
- [5] L. G. Bleris and M. V. Kothare, "Real-Time Implementation of Model Predictive Control," in *American Control Conference*, (Portland, OR), pp. 4166–4171, June 2005.
- [6] 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, 2002.
- [7] M. H. He and K. V. Ling, "Model Predictive Control on a Chip," in *The 5<sup>th</sup> International Conference on Control and Automation*, (Hungary, Budapest), pp. 528–531, June 26–29 2005.
- [8] E. E. Swartzlander and A. G. Alexopoulos, "The Sign/Logarithm Number System," *IEEE Transactions on Computers*, vol. 24, pp. 1238–1242, Dec. 1975.
- [9] J. G. Garcia, M. G. Arnold, L. G. Bleris, and M. V. Kothare, "LNS Architectures for Embedded Model Predictive Control Processors," in *2004 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, (Washington, DC), pp. 79–84, Sept. 2004.
- [10] M. G. Arnold, "An Improved Cotransformation for Logarithmic Subtraction," in *ISCAS'02*, (Scottsdale, Arizona), pp. 752–755, 26–29 May.
- [11] A. El-Amawy, "A Systolic Architecture for Fast Dense Matrix Inversion," *Transactions on Computers*, vol. 38, pp. 449–455, March 1989.
- [12] M. G. Arnold and J. Shuler, "A Processor that Converts Implicit Style Verilog into One-hot Designs," in *6<sup>th</sup> International Verilog HDL Conference*, (Santa Clara, CA), pp. 38–45, 1997. www.verilog.vito.com.
- [13] M. G. Arnold, *Verilog Digital Computer Design: Algorithms into Hardware*. NJ: PTR Prentice Hall, 1999.
- [14] www.adc.co.kr.