Proceedings of the 2006 American Control Conference
Minneapolis, Minnesota, USA, June 14-16, 2006

A Co-Processor FPGA Platform for the Implementation of Real-Time Model Predictive Control

Leonidas G. Bleris, Panagiotis D. Vouzis, Mark G. Arnold and Mayuresh V. Kothare

Abstract— **In order to effectively control nonlinear and multivariable models, and to incorporate constraints on system states, inputs and outputs (bounds, rate of change), a suitable (sometimes necessary) controller is Model Predictive Control (MPC). MPC is an optimization-based control scheme that requires abundant matrix operations for the calculation of the optimal control moves. In this work we propose a mixed software and hardware embedded MPC implementation. Using a codesign step and based on profiling results, we decompose the optimization algorithm into two parts: one that fits into a host processor and one that fits into a custom made unit, that performs the computationally demanding arithmetic operations. The profiling results and information on the co-processor design are provided.**

I. Introduction

Model Predictive Control (MPC) is broadly used in the chemical process industries, due to its ability to handle constraints and its applicability to multivariable nonlinear systems. It can be argued that today it is the single most commonly implemented advanced control algorithm, beyond PI/PID. MPC remains an open and growing area of research in systems and control. Nevertheless, because of the computational requirements of the optimizations associated with MPC, it can be applied to systems with slow dynamics. Furthermore, existing implementations of MPC typically perform numerical calculations using workstations in 64-bit floating-point arithmetic, which is too expensive, power demanding, and large in size, thus unsuitable for many application areas.

Recently, there has been considerable interest in expanding the applicability of MPC to other domains of engineering. In particular, attempts have been made to apply MPC to dynamical systems with fast response times and which were traditionally considered unsuitable for MPC implementation. A growing number of research groups is working on the subject during the last few years [1], [2]. Our group has examined alternative pathways for the implementation of MPC on-a-chip which are summarized in the following paragraphs.

Partial financial support for this research from the US National Science Foundation grant CTS-0134102 and the Pittsburgh Digital Greenhouse is gratefully acknowledged.

Leonidas G. Bleris is with the Department of Electrical and Computer Engineering, Lehigh University, Bethlehem, PA, 18015 bleris@lehigh.edu

Panagiotis D. Vouzis is with the Department of Computer Engineering, Lehigh University, Bethlehem, PA, 18015 vouzis@lehigh.edu

Mark G. Arnold is with the Department of Computer Science and Engineering, Lehigh University, Bethlehem, PA, 18015 maab@lehigh.edu

Mayuresh V. Kothare is with the Faculty of Chemical Engineering, Lehigh University, Bethlehem, PA, 18015 mayuresh.kothare@lehigh.edu

We have examined a general purpose processor implementation [3], [4]. We used a single board computer phyCORE-MPC555 that packs the power of Motorola's embedded 32-bit MPC555 microcontroller within a miniature footprint. The MPC555 is a high-speed 32-bit Central Processing Unit (CPU) that contains a 64-bit floating point unit designed to accelerate advanced algorithms. In order to implement the optimization required by MPC on this target, we used a combination of software tools: CodeWarrior Integrated Development Environment (IDE), MATLAB, Real-Time Workshop, and SIMULINK. By combining the MATLAB/SIMULINK environment with the Code-Warrior Development Studio, we were able to run Processor-In-the-Loop (PIL) simulations and analyze the performance using profiling techniques.

For an application-specific instruction processor implementation [5], [6], [7], [8] we proposed the following design framework. By emulating the microcontroller arithmetic operations, we reduce the precision of the microprocessor to the minimum, while maintaining stable control performance for a particular control application. This reduction is accomplished by a series of parametric tests using different word sizes and utilizing computational tools to simulate the controlled model. Taking advantage of the low precision, a Logarithmic Number System (LNS) based microprocessor architecture was used that provides energy and computational cost savings. This reduced-precision ASIP can achieve sampling speeds as low as 32msec for relatively large problems. Additionally, to quantify the advantage of reducing the precision, estimations for both 64-bit FP and 16-bit LNS circuits showed that for an arithmetic unit that computes addition, subtraction, multiplication and division, the size required is about 17 times larger for 64-bit FP.

In this paper we provide a mixed software-hardware embedded controller, extending the ideas presented in [9]. A codesign step is used prior to the actual implementation that decomposes the algorithm into two parts. One that fits into the host processor and one that fits into the custom made unit that performs all the (repetitive and computationally demanding) arithmetic operations. Thus the bulk of the MPC matrix calculations are performed in hardware and the rest in a general purpose microprocessor. The microprocessor acts as a master onto the co-processor by sending commands and data and receiving the results back, whenever necessary. Additionally, this architecture allows the microprocessor to perform other tasks while the co-processor is busy executing commands, and at the same time it improves the overall performance of the system. As will be described in the following sections, we selected the 16-bit Extensible Instruction Set

Controller (EISC) from ADCUS, Inc. as the microprocessor. For prototyping purposes, we use the ML401 board of Xilinx, that hosts a Virtex-IV Field Programmable Gate Array (FPGA). Both the microprocessor and coprocessor are synthesized and downloaded into the FPGA [10].

The paper is organized as follows: Section II provides a brief overview of MPC and the optimization algorithm used. Section III describes the rotating antenna case study, while Section IV contains the codesign analysis and the profiling results. In Section V, the proposed coprocessor architecture and related results are provided. The paper is summarized and concluded in Section VI.

II. Model Predictive Control

Controllers belonging to the MPC [11] family are generally characterized by the following steps: Initially the future outputs are calculated at each sample interval over a predetermined horizon N, the prediction horizon, using a process model. These outputs $y(t+k|t)$ for $k = 1...N$ depend up to the time t on the past inputs and on the future signals $u(t+k|t)$, $k = 0...N-1$ which are those to be sent to the system. The next step is to calculate the set of future control moves by optimizing a determined criterion, in order to keep the process as close as possible to a predefined reference trajectory. This criterion is usually a quadratic function of the difference between the predicted output signal and the reference trajectory. In some cases, in order to minimize the control effort the control moves $u(t+k|t)$ are included in the objective function:

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

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

where $y(t+k|t)$ are the predicted outputs, y_{ref} is the desired set reference output, $u(t+k|t)$ the control sequence and R is the weighting on the control moves, a design parameter. This system is subject to input constraints given by the vector b. Finally, the first control move $u(t|t)$ is sent to the system while the rest are rejected. At the next sampling instant the output $y(t+1)$ of the system is used in the optimization using feedback and the procedure is repeated so that we get an updated control sequence. The block diagram of MPC is depicted in Fig. 1.

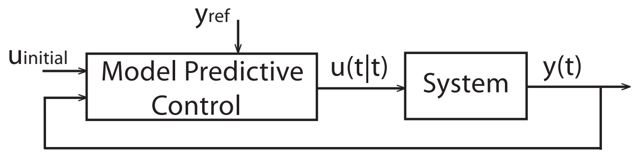

Fig. 1. System block diagram.

A. Optimization

The minimization of equation (1) subject to (2) that results in the optimal control moves $u(t+k|t)$ can be accomplished using several algorithms [12]. We use Newton's algorithm to solve the problem by incorporating the constraints in the cost function using barrier functions:

$$d_i(u) = \mu_i(a_i^T u - b_i)^2 \quad (3)$$

resulting in an unconstrained problem. The unconstrained problem can be solved numerically approximating J by a quadratic function around u, obtaining the gradient ∇J_P and Hessian H, and iterating:

$$u^{(k+1)} = u^{(k)} - H^{-1}(u^{(k)}) \cdot \nabla J_P(u^{(k)}) \quad (4)$$

III. Case Study: Antenna Rotation

The case study examined has been adapted from [13]. It is a state-space problem that describes the dynamics of a rotating antenna at the origin of the plane (driven by an electric motor). The control objective is to use the input voltage to the motor (u V) to rotate the antenna so that it always meets a predefined set-point (i.e. points towards a moving object in the plane). We assume that the angular positions of the antenna and the moving object (θ and θ_r rad respectively) and the angular velocity of the antenna ($\dot{\theta}$ rad/sec) are measurable. The motion of the antenna can be described by the following discrete-time equations obtained from their continuous-time counterparts by discretization using a sampling time of 0.1 s and Euler's first-order approximation for the derivative:

$$x(k+1) = \begin{bmatrix} 1 & 0.1 \\ 0 & 1-0.1a(k) \end{bmatrix} x(k) + \begin{bmatrix} 0 \\ 0.0787 \end{bmatrix} u(k)$$

$$y(k) = \begin{bmatrix} 1 & 0 \end{bmatrix} x(k)$$

The parameter $a(k)$ is proportional to the coefficient of viscous friction in the rotating parts of the antenna and for the simulations we assume that $a(k) = 1$.

The main complexity of MPC arises from the underlying optimization algorithm, and the size of the matrices involved. Therefore these can be made arbitrarily large by increasing the control and prediction horizons, allowing us to test the performance of the controller in different computational complexity levels. The MPC behavior was tested for different cases, having the set-point at 2 and the controlled variables constrained between $-2 < u < 2$ V. The output and the actuation for prediction horizon 20 and control horizons of 3,4,8 and 12 are illustrated in Fig. 2.

IV. Codesign

Hardware/Software (HW/SW) codesign refers to the simultaneous consideration of hardware and software within the design process [14], [15]. Codesign is becoming an increasingly more important research field primarily because of the advantages that it offers in terms of performance and flexibility of the resulting design.

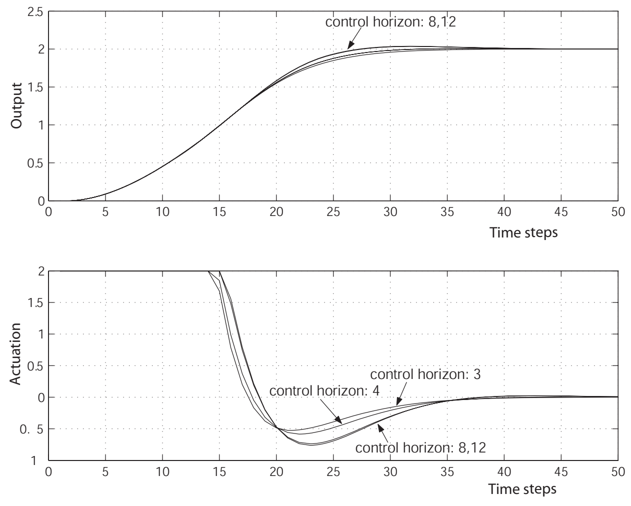

Fig. 2. Output and actuation for prediction horizon of 20 and control horizons of 3,4,8 and 12.

A general codesign flow is depicted in Fig. 3. The analysis starts with the system specification. The next step is the hardware and software partitioning phase. In this work we partition the algorithm to basic operational blocks, and analyze their behavior/performance using profiling techniques. Depending on the profiling results we assign these operational blocks to hardware or software. After partitioning the algorithm, the interface parts (communication and synchronization) between the hardware and software have to be designed. The interface will include all the communication mechanisms that permit the exchange of data between the processor that contains the software and the custom made hardware unit. Given the specifications on the hardware and software, the next step is the synthesis and compilation. A co-simulation step is then be used, where the custom unit is emulated together with the software. Finally during a verification process we ensure that the design is correct and complete [16], and that the design implements its specification accurately (e.g. performance constraints are met, cost of the design is acceptable).

A. Operational blocks of the optimization algorithm

The optimization algorithm consists of five functions which can be considered as five basic operational blocks. These functions are: the initializations prior to the optimizations, the calculation of the Gradient vector and the Hessian matrix, the Gauss-Jordan matrix inversion, and finally the optimal move calculation using Newton's iteration. For the particular case of using a state space model for the MPC formulation the outputs Y can be explicitly replaced in the cost function J by:

$$Y = A'X + B'U \tag{5}$$

where the prime denotes the adjusted A and B matrices [11], with a size depending on the control and prediction horizons. The Gradient and the Hessian used for the Newton's iteration

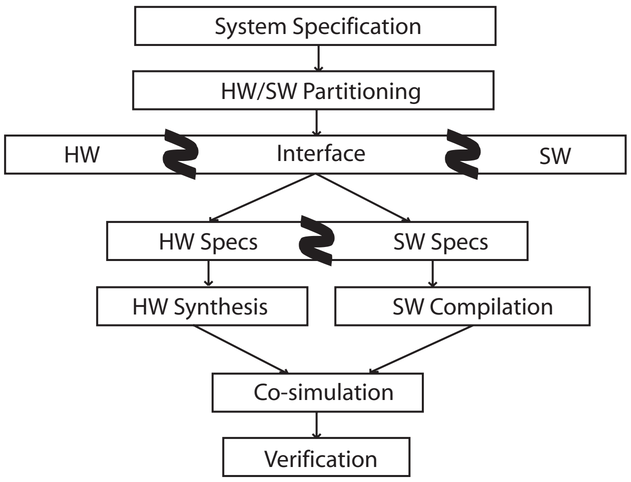

Fig. 3. Codesign Diagram.

(Eq. 4) are then given by the following formulas respectively:

$$\nabla J = 2B'^T B'U + 2B'^T A'x - 2B'^T Y_{ref} + 2RIU + B_{\nabla J} \tag{6}$$

where:

$$B_{\nabla J} = \mu \begin{bmatrix} -\frac{1}{(b+u_1)^2} + \frac{1}{(b-u_1)^2} \\ -\frac{1}{(b+u_2)^2} + \frac{1}{(b-u_2)^2} \\ \cdots \\ -\frac{1}{(b+u_m)^2} + \frac{1}{(b-u_m)^2} \end{bmatrix}$$

The Hessian matrix is given by:

$$H = 2B'^T B' + 2RI + \mu B_H \tag{7}$$

where B_H contains elements of the gradient of $B_{\nabla J}$ placed diagonally in a square matrix appropriately sized.

B. Profiling results

In order to partition the algorithm into hardware and software we examine the behavior of the operational blocks using a profiler. A profiler analyzes the amount of time a program spends performing these tasks and thus detects potential bottlenecks (time consuming routines that data passes through) and routines that are inordinately slow.

In Fig. 4 we present the profiling results of the five operational blocks, for a prediction horizon of 10 and variable control horizons. A direct observation is that the combined computational time for the calculation of the Gradient and the Hessian is approximately $70-80\%$ of the total optimization time. The next most expensive function is the matrix inversion which can take up to 30% of the total time, for large control horizons. Furthermore, we observe that for small control horizons the Gradient function uses almost half of the total optimization time and double the time of the Hessian function. Finally, we observe that by increasing the control horizon size, the matrix inversion becomes more expensive, and the computational time required by the Gradient and Hessian functions converge. The explanation is that with increasing control horizon the B_H becomes much more expensive than $B_{\nabla J}$. In Fig. 5 we present the profiling results

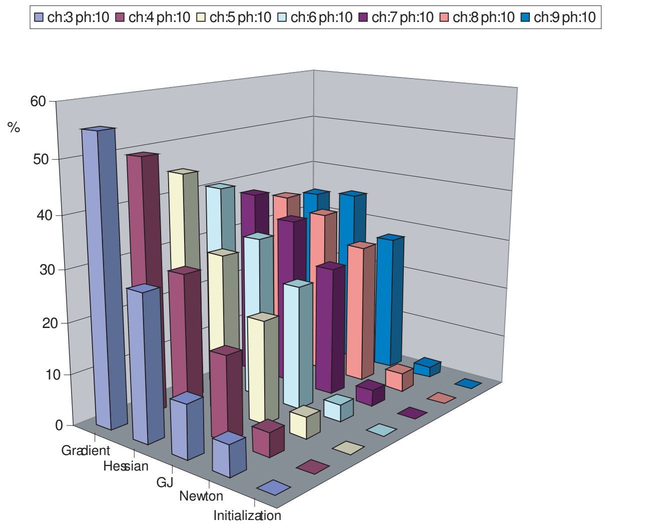

Fig. 4. Profiling results for prediction horizon 10 and variable control horizon.

of the five operational blocks, for a prediction horizon of 20 and variable control horizons. The results are in accordance with the previous profiling results (Fig. 4). One additional observation is the influence of the size of the prediction horizon on the computational complexity of the Gradient function (for small control horizons). In Fig. 6 we present

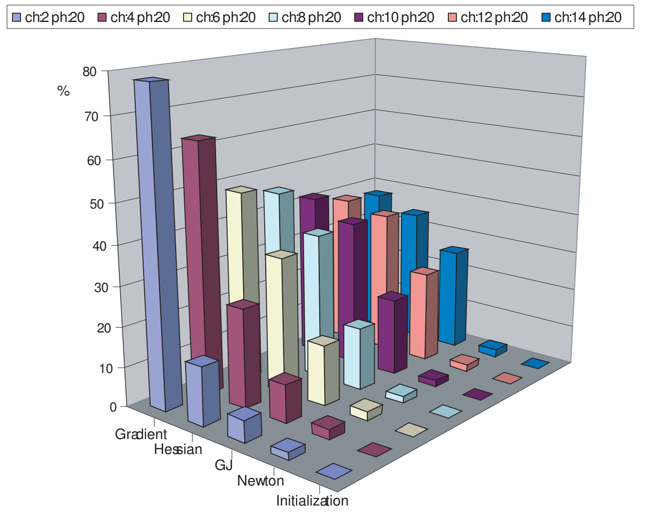

Fig. 5. Profiling results for prediction horizon 20 and variable control horizon.

the cumulative time spent for the profiling, for a prediction horizon of 20 and variable control horizons. Note that we run the optimization for a fixed number of 3000 iterations. This is unnecessarily large for the calculation of the optimal control moves, but allows for accurate profiling results. In Fig. 7 we present the profiling results for control horizon 6, prediction horizon 10 and variable number of optimizations.

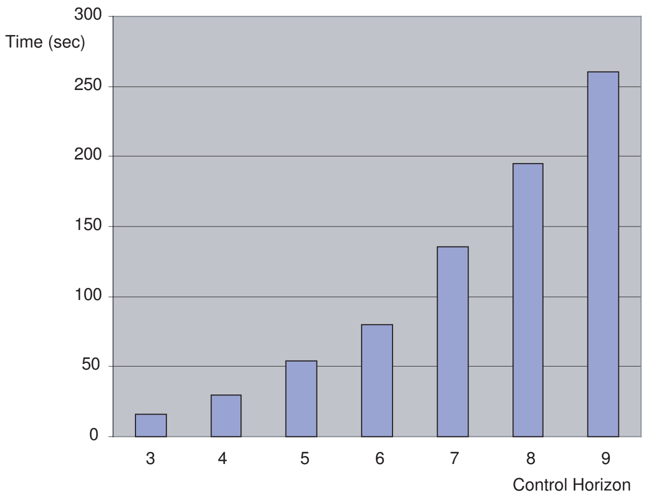

Fig. 6. Cumulative time spend for the profiling, for a prediction horizon of 20 and variable control horizons.

As expected the number of iterations in the optimization does not influence the distribution of the computational time of the five basic operational blocks.

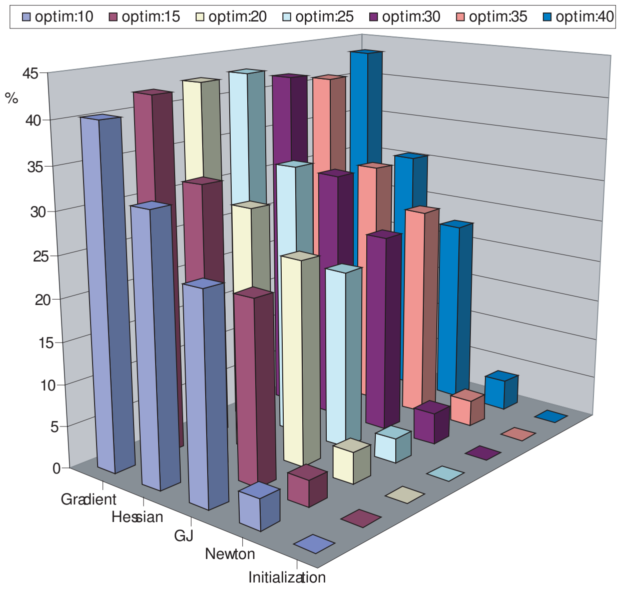

Fig. 7. Profiling results for control horizon 6, prediction horizon 10 and variable number of optimizations.

V. Co-processor design

This higher level analysis of the MPC optimization code reveals that the repetitive matrix operations of the Gradient and Hessian, are responsible for the major part of the processing. Therefore these specific matrix operations have to be implemented efficiently, while the rest of the operations (initializations, Newton's iteration) can be performed using a general purpose microprocessor.

The solution to this problem can be given using a limited precision host microcontroller together with a matrix processor that acts as a hardware accelerator for the matrix operations. The selected microprocessor that acts as the host for our design is the 16-bit Extensible Instruction Set Computer (EISC) from ADCUS [17] and for prototyping we use the FPGA Virtex-IV XC4VLS25 device of Xilinx. The system is interfaced with MATLAB, running on a workstation, in order to allow PIL simulations for testing and debugging purposes. Both the ADCUS microprocessor and the matrix co-processor are described in Verilog and the whole design is synthesized with the ISE 7.1 design environment of Xilinx.

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. This configuration is illustrated in Fig.8.

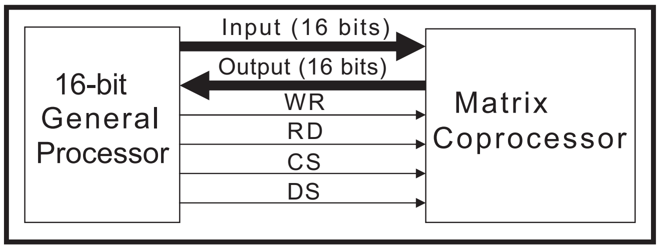

Fig. 8. The general architecture of the microprocessor-coprocessor system.

B. Datapath

The matrix processor consists of one-hot controller and associated datapath generated by a tool called VITO [18], [19]. 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; usually the case in FPGA devices. The coprocessor datapath includes two matrix registers, **A** and **C**, each of which contains an $r \times c$ matrix where $r, c \leq 2^m$, and the constant m is the size of the index in bits used to address the matrices. Initially, the host sends r, c to the matrix coprocessor, which then defines the size of the matrices needed to process the MPC model for a particular control application. **A** and **C** both are 16×2^{2m} bit memories. **A** is a dual-port memory, where one of its ports is attached to a pipelined LNS ALU of the kind described in [20]. 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. 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 $0 \leq i, j, k < 2^m$.

In addition to the matrix registers, **A** and **C**, the processor has a main memory, `m[ ]`, used to store the several matrices needed by MPC. When the host requests the matrix processor to perform a command (e.g. 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 $r \times c$ matrix takes $r \cdot c$ 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 inadequate for a particularly large MPC problem, commands are provided to transfer $r \cdot c$ words at the maximum rate supported by the host interface.

Table I illustrates some of the operations supported by the matrix processor; "host" indicates a data transfer to/from the host, i and j are indices provided by the host; $\mathbf{A}_i$ and $\mathbf{C}_i$ are rows chosen by the host; `i` is an internal register; **b** and **B** are vectors and matrices, respectively; (stored in `m[ ]` at the base address specified by the host); $\mathbf{B}^T$ is a transpose, **I** is the identity matrix; **0** is the zero matrix; p is an index register that indicates the pivot row (as needed by matrix inversion methods, e.g. Gauss-Jordan); x is an LNS scalar value.

The coprocessor's instruction set simplifies considerably the software development for the MPC algorithm. For example, a matrix multiplication between two matrices A,B, in the C programming language, would be:

```
for (i = 0; i < M; i++)
  for (j = 0; j< M; j++){
    A[i][j]=0.0;
      for (k = 0; k < P; k++)
        A[i][j] += B[k][i]*C[k][j];
  }
```

The same operation is reduced to the following two commands:

```
CoProcessor=COMMAND_MUL_C;
CoProcessor=Memory_location;
```

where the coprocessor receives the instruction to perform a multiplication between the matrix register **C** and the matrix **B**, with base address `Memory_location`. While the coprocessor is busy executing this command, the microprocessor can execute alternative tasks.

After the synthesis, the microprocessor and the coprocessor occupy 2943 and 1908 slices of the FPGA respectively, and the whole systems works at 25MHz. As a case study we used the antenna control problem described in Section III. For a control horizon of 3 and a prediction horizon of 10 the optimization problem is solved within 0.89ms. The same control problem solved by using Motorola's 32-bit MPC555 processor, running at 40MHz and incorporating a double precision FP unit, requires 15ms for the same control and prediction horizons [3]. Analytical synthesis and timing results of the proposed architecture are presented in [10] for two different case studies, the antenna rotation and a glucose regulation problem.

Mnem.	$O(n)$ Oper.	Mnem.	$O(n^2)$ Oper.	Mnem.	$O(1)$ Oper.
STOREVA	$\mathbf{b} \leftarrow \mathbf{A}_i$, $i < n$	INPC	$\mathbf{C} \leftarrow$ host	PUTN	$n \leftarrow$ host
	$\mathbf{b} \leftarrow \mathbf{A}_p$, $i = n$	OUTA	host $\leftarrow \mathbf{A}$	PUTA	$\mathbf{A}_{i,j} \leftarrow$ host
STOREVC	$\mathbf{b} \leftarrow \mathbf{C}_i$, $i < n$	OUTC	host $\leftarrow \mathbf{C}$	PUTC	$\mathbf{C}_{i,j} \leftarrow$ host
	$\mathbf{b} \leftarrow \mathbf{C}_p$, $i = n$	LOADA	$\mathbf{A} \leftarrow \mathbf{B}$	PUTP	$p \leftarrow$ host
LOADVA	$\mathbf{A}_i \leftarrow \mathbf{b}$, $i < n$	LOADC	$\mathbf{C} \leftarrow \mathbf{B}$	PUTX	$x \leftarrow$ host
	$\mathbf{A}_p \leftarrow \mathbf{b}$, $i = n$	LOADCT	$\mathbf{C} \leftarrow \mathbf{B}^T$	GETA	host $\leftarrow \mathbf{A}_{i,j}$
LOADVC	$\mathbf{C}_i \leftarrow \mathbf{b}$, $i < n$	STOREC	$\mathbf{B} \leftarrow \mathbf{C}$	GETC	host $\leftarrow \mathbf{C}_{i,j}$
	$\mathbf{C}_p \leftarrow \mathbf{b}$, $i = n$	ADD	$\mathbf{A} \leftarrow \mathbf{A} + \mathbf{B}$	GETP	host $\leftarrow p$
ADDXVA	$\mathbf{A}_i \leftarrow \mathbf{A}_i + x \cdot \mathbf{b}$, $i < n$	MULV	$\mathbf{A}_i \leftarrow \mathbf{C}\mathbf{b}$	GETX	host $\leftarrow x$
	$\mathbf{A}_p \leftarrow \mathbf{A}_p + x \cdot \mathbf{b}$, $i = n$	STOREAZ	$\mathbf{B} \leftarrow \mathbf{A}$		
ADDXVC	$\mathbf{C}_i \leftarrow \mathbf{C}_i + x \cdot \mathbf{b}$, $i < n$		$\mathbf{A} \leftarrow \mathbf{0}$		
	$\mathbf{C}_p \leftarrow \mathbf{C}_p + x \cdot \mathbf{b}$, $i = n$	STOREAI	$\mathbf{B} \leftarrow \mathbf{A}$		
PIVOT	$x \leftarrow \max(\lvert\mathbf{C}_{i,j}\rvert, ..., \lvert\mathbf{C}_{n-1,j}\rvert)$		$\mathbf{A} \leftarrow \mathbf{I}$	Mnem.	$O(n^3)$ Oper.
	$p \leftarrow \mathtt{i}$ such that $C_{\mathtt{i},j} = x$			MUL	$\mathbf{A} \leftarrow \mathbf{C}\mathbf{B}$

TABLE I

MATRIX PROCESSOR OPERATIONS LISTED BY SINGLE-ALU DELAY.

VI. CONCLUSIONS

In this work we have provided research results of the implementation of a FPGA coprocessor for real-time MPC applications. There is a spectrum of possible implementations for embedded MPC ranging from software to complete hardware. While a pure software approach can be more direct, a hardware-software codesign approach leading to an FPGA implementation can be an efficient solution when performance, reduced memory, and low-power consumption are sought.

The advantages of MPC such as the ability to handle constraints, the applicability to nonlinear processes and to multivariable problems, make this control method a necessary choice for many control problems. For example, there are numerous potential applications for biomedical control including: control of physiological processes, glucose control, drug infusion control, cardiac pacemakers and defibrillators, blood flow and pressure control, and neurological implants. On the other hand, several small-scale industrial application areas that have fast dynamics will benefit from a real-time implementation of MPC. These areas include MEMS, automotive/aerospace control, power electronics, and microchemical systems.

REFERENCES

[1] K.V. Ling, S.P. Yue, and J.M. Maciejowski. An FPGA Implementation of Model Predictive Control. In *Americal Control Conference*, Minneapolis, MN, June 2006.

[2] T. A. Johansen, W. Jackson, R. Schreiber, and P. Tondel. Hardware Architecture Design for Explicit Model Predictive Control. In *2006 American Control Conference*, Minneapolis, MI, July 2006.

[3] L. G. Bleris and M. V. Kothare. Real-time implementation of model predictive control. In *2005 American Control Conference*, pages 4166–4171, Portland, OR, July 2005.

[4] L. G. Bleris and M. V. Kothare. Implementation of Model Predictive Control for Glucose Regulation using a General Purpose Microprocessor. In *44th IEEE Conference on Decision and Control and European Control Conference*, Seville, Spain, December 2005.

[5] L. G. Bleris, M. V. Kothare, J. G. Garcia, and M. G. Arnold. Embedded model predictive control for system-on-a-chip applications. In *Proceedings of the 7^{th} IFAC Symposium on Dynamics and Control of Process Systems (DYCOPS-7)*, Boston, MA, July 2004.

[6] 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*, pages 79 – 84, Washington, D.C., September 2004.

[7] L. G. Bleris, M. V. Kothare, J. G. Garcia, and M. G. Arnold. Towards embedded model predictive control for system-on-a-chip applications. *Journal of Process Control*, 16:255–264, 2006.

[8] L. G. Bleris, J. G. Garcia, and M. V. Kothare. Model predictive hydrodynamic regulation of microflows. In *2005 American Control Conference*, pages 1752–1757, Portland, OR, July 2005.

[9] P. Vouzis, L. G. Bleris, M. V. Kothare, and M. G. Arnold. Towards a Co-design Implementation of a System for Model Predictive Control. In *2005 AIChE Annual Meeting*, Cincinnati, OH, November 2005.

[10] P. Vouzis, L. G. Bleris, M. V. Kothare, and M. G. Arnold. A system-on-a-chip implementation for embedded real-time model predictive control. *Submitted: IEEE Transactions on Control Systems Technology*, 2006.

[11] E. F. Camacho and C. Bordons. *Model Predictive Control*. Springer, New York, 1999.

[12] R. Fletcher. *Practical Methods of Optimization*. John Wiley & Sons, 1987.

[13] M. V. Kothare, V. Balakrishnan, and M. Morari. Robust constrained model predictive control using linear matrix inequalities. *Automatica*, 32(10):1361–1379, October 1996.

[14] S. Edwards, L. Lavagno, E. A. Lee, and A.Sangiovanni-Vincentelli. Design of embedded systems: formal models, validation, and synthesis. *Proceedings of the IEEE*, 85:366–390, March 1997.

[15] M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, and A. Sangiovanni-Vincentelli. Hardware/software codesign of embedded systems. *IEEE Micro*, 14:26–36, August 1994.

[16] F. Vahid and T. Givargis. *Embedded System Design: A Unified Hardware/Software Introduction*. John Wiley & Sons, Inc, New York, NY, 2002.

[17] www.adc.co.kr.

[18] M. Arnold and J. Shuler. A preprocessor that converts implicit style verilog into one-hot designs. In *6th International Verilog HDL Conference*, pages 38–45, Santa Clara, CA, March 31 - April 3 1997. For more recent versions see: ` `www.verilog.vito.com`.

[19] M. G. Arnold. *Verilog Digital Computer Design: Algorithms into Hardware*. PTR Prentice Hall, Upper Saddle River, NJ, 1999.

[20] M. G. Arnold. Improved Cotransformation For LNS Subtraction. *IEEE International Symposium on Circuits and Systems*, II:752–755, May 2002.