# Improving the Performance of Active Set Based Model Predictive Controls by Dataflow Methods

Ruirui Gu, *member IEEE*, Shuvra S. Bhattacharyya, *senior member IEEE* and William S. Levine, *fellow IEEE*

*Abstract*— Dataflow representations of Digital Signal Processing (DSP) software have been developing since the 1980's. They have proven to be useful in identifying bottlenecks in DSP algorithms, improving the efficiency of the computations, and in designing appropriate hardware for implementing the algorithms. This paper demonstrates the use of dataflow to improve a Model Predictive Control (MPC) algorithm. MPC has been extensively used in the real world but its application has been limited to relatively slow processes because it is computationally intensive. It is shown that the time required for the MPC computations using a representative active set method for solving the optimization problem can be reduced by means of dataflow analysis and implementation improvements based on these analyses.

## I. INTRODUCTION

In an earlier paper [1], we used dataflow techniques to improve the speed of MPC control algorithms that utilize the Newton-KKT method to solve for the control value. Newton-KKT is an example of an interior-point method for solving Quadratic Programming (QP) problems. Many people prefer the so-called active-set methods [2] for solving these problems. In this paper, we apply similar but different dataflow techniques to improve the performance of a representative active set method.

The motivation to the work is that although MPC has been applied to many practical problems, it is computationally intensive. Because the computation must be completed in a limited time interval, MPC has been limited to relatively slow processes.

As a result, there has been considerable research aimed at speeding up the computation of optimal controls for MPC. Most of this research has concentrated on improving the algorithms. Relatively little work [3] has been devoted to improving the implementation of the algorithms. But the two go hand in hand. A specific example that is especially relevant to the work reported here is that an algorithm that can be executed with many parallel steps will be much faster if properly implemented than a more efficient algorithm that must operate sequentially. This example is meaningful especially due to fast development of the multi-core parallel processor.

R. Gu is with the Department of Electrical and Computer Engineering, and Institute for Advanced Computer Studies, University of Maryland, College Park, MD, 20742. `rgu@umd.edu`

S. S. Bhattacharyya is with the Department of Electrical and Computer Engineering, and Institute for Advanced Computer Studies, University of Maryland, College Park, MD, 20742. `ssb@umd.edu`

W. S. Levine is with the Department of Electrical and Computer Engineering, University of Maryland, College Park, MD, 20742. `wsl@umd.edu`

The paper is organized as follows. Because related work in both controls and computer engineering was summarized in our earlier paper [1], and space is limited we omit such a survey here. We describe the dataflow framework to be applied to MPC in the following section. We then illustrate profiling and the possible methods which can be used to improve the MPC system performance. Finally, we apply our methods to an application of Maciejowski et al. [4].

## II. QUADRATIC PROGRAMMING AND THE RCDF MODEL

### A. Quadratic Programming and Solutions

In MPC, the controller is often designed to solve a quadratic programming (QP) problem, which is used to find the optimized input. Consider the following Quadratic Programming (QP) problem in standard inequality form $< P >$:

$$minimize \frac{1}{2} < x, Qx > + < c, x >,$$

s.t.

$$a_i^T x \leq b_i, x \in R^n, i \in [1, 2, \ldots, m] \equiv I$$

$$x_{min} \leq x \leq x_{max},$$

with $Q \in R^{n \times n}$ symmetric, $c \in R^n$, $a_i \in R^n$, $b_i \in R$, and $x_{min}, x_{max} \in R^n$. Sometimes there are optional equality constraints in the form of

$$a_i^T x = b_i, i \in [1, 2, \ldots, r] \equiv J.$$

Usually the initial point $x_0$ is set in the feasible range.

The objective function is minimized at each sampling interval, in order to solve for an optimal open loop control trajectory over that horizon. Either an active set method or an interior point method is a good candidate to solve the QP problem. They each have advantages in different situations. Interior point methods usually require fewer iterations than the active set methods. But a larger number of variables is necessary in order to solve by active set methods. For either interior point or active set methods, intensive computations are required. For both methods, an excessive number of iterations is sometimes possible. Many variables and a large number of iterations result in computational complexity, and in turn result in delay of system execution. For real time applications, if the delay is bigger than the threshold of the sampling interval, MPC is not feasible.

## B. RCDF Model

In this paper, for MPC systems, we use a new dynamic dataflow modeling technique, called reactive, control-integrated dataflow (RCDF). This approach is more specialized than other dynamic dataflow techniques, but for MPC, this specialization can be exploited in useful ways to streamline the implementation process.

We need very little of the full RCDF formalism. It is mainly necessary to understand that for DSP-oriented dataflow graphs, vertices (*actors*) represent computations of arbitrary complexity, and an edge represents the flow of data as values are passed from the output of one computation to the input of another. Each data value is encapsulated in an object called a *token* as it is passed across an edge. Actors are assumed to execute iteratively, over and over again, as the graph processes data from one or more data streams. These data streams are typically assumed to be of unbounded length (e.g., derived implementations are not dependent on any predefined duration for the input signals). In dataflow graphs, interfaces to input data streams are typically represented as *source* actors (actors that have no input edges).

## C. Applying An RCDF Model to Active Set Method

The dataflow framework provides a complete solution from system modeling to optimized implementation, as shown in Figure 1. First of all, the control algorithm is modeled as a dataflow model. After all the computation tasks are divided into different actors, we profile the execution time of each actor to determine the bottleneck(s) of the system performance. We then use DIF to look for solutions for different actors.



Fig. 1. Dataflow Framework for efficient system implementation

## D. The Active Set Method in the Simulation

The active set method used in this paper is from MATLAB. The M-file *quadprog.m* is designed for quadratic programming (QP), and the private function *qpsub.m* is especially to solve QP problems using an active set method. The algorithm is similar to the description in [5]. We do not claim that *qpsub.m* is a particularly good algorithm. The intent is primarily to demonstrate and test the RCDF approach to improving the implementation of an active set algorithm. Finding the best active set algorithm is a separate issue.

First of all, a feasible starting point is computed by either a Phase I approach or the big M method. Since computation of a feasible point is done only once for a QP problem, the burden from this computation usually is not heavy. In the $k$th iteration, inequality constraints are partitioned into two sets: active (or sufficiently close to be deemed active for this iteration) and inactive. The active set for one iteration is sometimes called the working set. We define the working set $W_k$ at iteration $k$ for current state variable $x_k$ as:

$$W_k(x_k) = \{i \in I \bigcup J : a_i^T x_k = b_i\}.$$

Given an iteration $x_k$ and the current active set, we check whether $x_k$ minimize the objective function in the subspace defined by the active set. If not, then we compute the search direction $s_k$ starting from the current $x_k$. Next, it is required to decide how far to move along this direction. We update the current value of $x_k$ using $x_k = x_k + \alpha_k s_k$. If $\alpha_k = 1$, there are no new constraints active at $x_k + \alpha_k s_k$, and there are no blocking constraints. Otherwise, if $\alpha_k < 1$, that is, the step along $s_k$ was blocked by some constraints not in $W_k$, then a new working set $W_{k+1}$ is constructed by adding one of the blocking constraints to $W_k$.

## E. Dataflow Model of Active Set Method

In [1], we have applied the RCDF model to one kind of interior point method, the Newton-KKT method [6]. In this paper, we model and optimize the active set method described in section II-D. The original system model of the active set method is described in an RCDF model, as shown in Figure 2. The functionality of actors is described as follows:
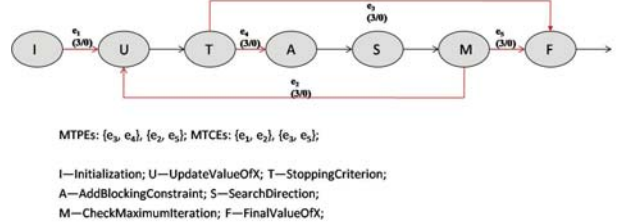


MTPEs: {e₃, e₄}, {e₂, e₅}; MTCEs: {e₁, e₂}, {e₃, e₅};

I—Initialization; U—UpdateValueOfX; T—StoppingCriterion;
A—AddBlockingConstraint; S—SearchDirection;
M—CheckMaximumIteration; F—FinalValueOfX;

Fig. 2. The original RCDF model of the active set method.

$I$—The actor $I$ is used to initialize the values of state variables and the values of parameters, such as the tolerance threshold, that are used later. The initial point $x_0$ in the problem $P$ is either set manually, or computed in some way to make sure the point is feasible. In this paper, we use a Phase I approach to compute a feasible initial point.

$U$—The actor $U$ is used to compute the updated values of $x$. In the beginning of an iteration, the token is passed from actor $I$, and then the token is passed from actor $M$, depending on stopping criterion and iteration number.

$T$—The actor $T$ is used to check if a minimum is reached in the current subspace. Here the stopping criterion is also checked.

$A$—The actor $A$ is used to add blocking constraints to the working set. As we said in section II-D, if the step length was blocked by some constraints not in the current active set, we add one of the blocking constraints to construct a new working set. If there is no chance of cycling then we pick the constraint which causes the biggest reduction in the cost function,i.e the constraint with the most negative Lagrange multiplier.

$S$—The actor $S$ is used to compute the search direction for the next iteration. It finds the solution by solving a linear system of equations.

| $ncond$ | 3 | | 6 | | 9 | |
|---|---|---|---|---|---|---|
| statistics | mean | variance | mean | variance | mean | variance |
| I | 0.003 | 3e-5 | 0.001 | 2e-6 | 0.001 | 4e-6 |
| U | 0.016 | 5e-6 | 0.002 | 1e-5 | 0.002 | 7e-6 |
| T | 0.031 | 9e-6 | 0.005 | 2e-5 | 0.004 | 5e-6 |
| A | 0.141 | 3e-5 | 0.125 | 3e-5 | 0.203 | 5e-5 |
| S | 0.078 | 2e-5 | 0.088 | 2e-5 | 0.047 | 1e-5 |

$M$—The actor $M$ is used to check the number of iterations. If the maximum is reached, the loop is terminated.

$F$—The actor $F$ is used to record the optimized value of $x$.

Based on our dataflow-based modeling approach, along with MATLAB implementations of the individual actors, we have conducted MATLAB simulations to evaluate the contribution of each actor to the overall execution time required for the application.

## III. ANALYSIS AND IMPROVEMENT

### A. Profiling

In our MATLAB simulations, the matrix $Q$ was generated based on the condition number ($ncond$), and the number of negative eigenvalues ($ngeig$). We fixed the parameter $ngeig$ to be 0 with $Q > 0$ enforced, and chose $ncond$ from the set $\{3, 6, 9\}$. It is noted that before profiling we have ignored certain "fine-grained" actors that have very low computational cost. For example, we have ignored actor $M$, which is used only to check if the iteration number has exceeded the maximum. We have also ignored the execution time contributions of actor $F$, which involves a simple operation to record the final optimized value.

Table I shows the execution times of different actors for different values of $ncond$ and randomly generated $Q$. The units for the numbers in Table I are seconds. These values were determined by implementing each actor in MATLAB (version 7.04), executing the code for each 100 by 100 matrix $Q$, and recording the mean and variance of the time required to complete the computations.

According to the statistical data in Table I, the computational burden from the listed actors can be ordered as follows:

$$b_A > b_S > b_T > b_U > b_I,$$

where $b_{Actor}$ indicates the computational burden from the actor. The bottlenecks are the actors with relatively larger computational burden, and in this system, we address the actors $A$ and $S$.

In the next section, we describe how our dataflow-based model together with its profiled execution time information can be used to strategically dedicate parallel hardware resources and accelerate the computation of performance-critical components in the overall design.

### B. Improvement from Data Parallelism

Since the algorithm is decomposed into actors based on functionality, the code size and complexity generally varies across the actors. This is typical of dataflow-based program representations. Some of the more complex actors, most notably $A$, represent *hierarchical actors* whose internal functionalities are described by additional ("nested") dataflow graphs. We elaborate on the internal representation of actor $A$ below.

To alleviate the bottleneck due to $A$, we examined the MATLAB source code for $A$ and replaced the actor $A$ in Figure 2 with the equivalent, hierarchical ("nested") RCDF subgraph shown in Figure 3. The detailed description of each actor is shown in Figure 4.
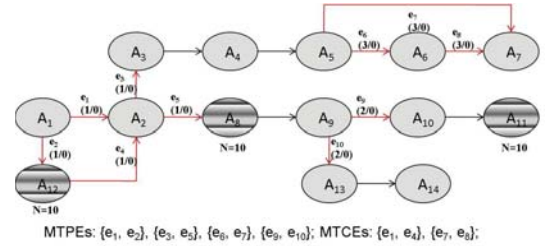


MTPEs: {$e_1$, $e_2$}, {$e_3$, $e_5$}, {$e_6$, $e_7$}, {$e_9$, $e_{10}$}; MTCEs: {$e_1$, $e_4$}, {$e_7$, $e_8$};

Fig. 3.   RCDF model of super actor A.

$A_1$—CheckIfHitConstraints; $A_2$—CheckIfSimplexIteration;
$A_3$—ComputeSize; $A_4$—ComputeValueOfZ;
$A_5$—CheckNumberOfVariables; $A_6$—UpdateFlagSignal;
$A_7$—ResetParameterValue; $A_8$—FindDataIndexInRange;
$A_9$—CheckValueOfLength; $A_{10}$—UpdateParametersValue;
$A_{11}$—DeleteColumnFromMatrix; $A_{12}$—InsertColumnIntoMatrix;
$A_{13}$—ComputerValueOfLambda; $A_{14}$—UpdateValueOfACTIND;

Fig. 4.   Description of subgraph for actor A.



MTPEs: {$e_3$, $e_4$}, {$e_2$, $e_5$}; MTCEs: {$e_1$, $e_2$}, {$e_3$, $e_5$}; {$e_6$, $e_7$};

I—Initialization; U—UpdateValueOfX; T—StoppingCriterion;
A—AddBlockingConstraint; $S_1$—SearchDirectionOption1;
$S_2$—SearchDirectionOption2; R—MVOS;
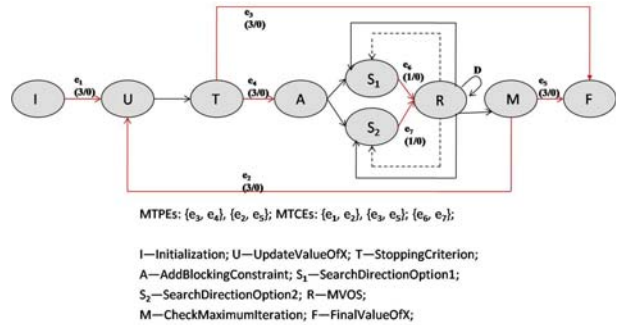M—CheckMaximumIteration; F—FinalValueOfX;

Fig. 5.   RCDF model of active set method after application of multi-version transformations.

Data parallelism exposes concurrency across sets of data to which the same operation is to be applied. Suppose there is a group of tokens indicated as $t_1, t_2, \ldots, t_n$ and there is a computational task $C$ that can be applied to each token independently. Here, "independently" means execution of $C$ on any given $t_i, 1 \leq i \leq n$ neither depends on nor

| $ncond$ | system before DPT | | system after DPT | |
|---|---|---|---|---|
| statistics | mean | variance | mean | variance |
| 0 | 0.36406 | 0.001 | 0.35156 | 0.000 |
| 3 | 0.40156 | 0.03 | 0.40080 | 0.001 |
| 6 | 0.39531 | 0.001 | 0.38935 | 0.000 |
| 9 | 0.43906 | 0.007 | 0.41375 | 0.001 |
| 12 | 0.42031 | 0.001 | 0.40563 | 0.000 |

affects any other token $t_j, j \neq i$. In this case, if we apply $C$ to each token $t_1, t_2, \ldots, t_n$ separately at the same time, then the total execution time for processing these tokens is reduced by a factor of $n$ compared to the time required by sequential processing (assuming that each token requires the same amount of processing time by $C$).

Exploiting data parallelism in this way can be viewed as a transformation on the given dataflow graph. We refer to this transformation as the *data parallelism transformation* (DPT).

In terms of hardware implementation, data parallel operation (the DPT) requires the use of parallel processing units. Since the computation for each token is independent, the synchronization overhead among the processing units is minimal. The fastest case arises when the number of parallel processing units is large enough so that each token can be mapped to a separate processing unit. In case the number of processing units is limited, windows of tokens are dispatched for parallel processing, and any unprocessed tokens outside the window are queued for later dispatching.

In the system shown in Figure 3, we can identify the actors $A_8$, $A_{11}$ and $A_{12}$ as candidates for exploiting data parallel operation. These actors are indicated as shaded actors in Figure 3. The original MATLAB code does not take advantage of data parallelism inside any of them. By windowing across successive groups of tokens, it is possible to apply the DPT to all of these actors. Inside each actor $A_8$, $A_{11}$ and $A_{12}$, the original system deals with all tokens in a sequential way, as shown in the left side of Figure 6. After applying the DPT to modify the implementation, parallel processing units conduct concurrent groups of independent computations, as shown in the right side of Figure 6. For one actor, we define the *DPT degree* as the number of tokens that are processed in parallel. In Figure 6, the DPT degree is 3.



Fig. 6.   Example of execution before and after the application of DPT.

The performance improvement due to our application of the DPT is shown as Table II. In this experiment, we have applied the DPT to the following actors: $A_8$, $A_{11}$, $A_{12}$, and for each actor, we have used $N = 10$ as the *DPT degree*.

## C. Improvement from Multi-version Parallelism

Besides data parallelism, the other important form of parallelism exposed effectively by dataflow graphs is functional parallelism. Functional parallelism refers to the simultaneous execution of distinct actors on separate hardware resources, whereas data parallelism involves simultaneously executing the same actor on separate resources.

A hybrid form of parallelism, which we call *multi-version parallelism*(MVP), can be very useful when hardware resources are relatively abundant and constraints on performance are relatively stringent. We view multi-version parallelism as a hybrid form because it relates to aspects of both functional and data parallelism.

Next we solve the bottleneck due to actor $S$ by taking advantage of functional parallelism. In the simulation, we use two different search algorithms for determining the search direction: Newton's method denoted by actor $S_1$ and steepest decent denoted by actor $S_2$.

The time required to complete an execution of $S_1$ or $S_2$ is in general data-dependent, and the relative speeds of corresponding executions (i.e., executions that have the same "$j$" index) are also data-dependent. In general, for some values $j \in J_1$, each $j$th execution of $S_1$ will complete before the $j$th execution of $S_2$, and for other values $j \in J_2$ ($J_1 \cap J_2 = \emptyset$), the $j$th executions of $S_2$ will complete sooner.

A multi-version implementation of the search direction calculation based on alternative implementations $S_1$ and $S_2$ therefore involves executing them both in parallel (simultaneously on separate resources), and taking the result of the $S_i$ that finishes first. As soon as one of the "versions" completes, its result is taken as the result of the corresponding execution of the search direction calculation, and the current execution of the other version is terminated. Such a multi-version implementation is useful whenever there can be significant variation between which of the versions completes first, and the available hardware resources accommodate parallel execution of the different versions.

The modified system with MVP is shown in Figure 5. Actor $R$ in Figure 5 represents a special actor, which we call a *multi-version output selector* (*MVOS*), for multi-version implementation. $R$ is an RCDF actor that samples its input edges in some pre-defined order. As soon as it finds an input edge that contains a token, it reads the token and copies it to its output. The actor then samples the remaining inputs and discards any tokens that it finds on those inputs — this happens in the event that different versions of the associated multi-version actor have produced their outputs at relatively closely-spaced points in time. After any such "redundant" inputs have been discarded, an MVOS actor sends a single token to each of the separate "version" actors ($H_1$ and $H_2$ in this example) to enable the next invocations of these actors. The use of such enabling tokens ensures that at most one execution of each version actor is allowed to execute at any given time (i.e., version-level, data-parallel operation is prevented). Use of these enabling tokens can be "optimized away" if other details in the implementation preclude data

TABLE III

SIMULATION RESULTS OF SYSTEM WITH AND WITHOUT MVP.

| ncond | option1 | | option2 | | MVP system | |
|---|---|---|---|---|---|---|
| statistics | mean | variance | mean | variance | mean | variance |
| 0 | 0.350 | 0.000 | 0.252 | 0.001 | 0.252 | 0.000 |
| 3 | 0.402 | 0.002 | 0.450 | 0.004 | 0.401 | 0.001 |
| 6 | 0.381 | 0.001 | 0.323 | 0.001 | 0.323 | 0.001 |
| 9 | 0.438 | 0.013 | 0.428 | 0.001 | 0.428 | 0.000 |
| 12 | 0.402 | 0.008 | 0.455 | 0.002 | 0.402 | 0.008 |

parallel operation — for example, if each version actor is mapped to a single hardware resource that is capable of performing only one version execution at any given time.

The MVOS can also optionally send an asynchronous reset signal to each version actor. The asynchronous reset signals generated by the MVOS actor $R$ are represented by dashed lines in Figure 5. For more details of asynchronous reset signals refer to [1].

The *self loop* edge of $R$ (the edge whose source and sink are both $R$) represents the state variable used by $R$ that determines whether 1) the actor is presently monitoring its input edges to determine which version has won the race for the current execution, or 2) the actor is in a state of discarding input tokens because the race is over. The $D$ next to this edge represents a unit *delay*. Such delays represent initial tokens on edges.

When we apply MVP to the original system, the system performance improvement is shown as Table III. Besides improving system performance, MVP also plays an important role when handling failed operation, which is critical to real-time system. For the details of the handling mechanism refer to [1].
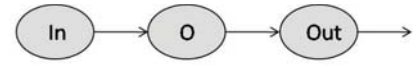
### D. Hierarchical System Model

RCDF specifications, as with most other forms of dataflow specifications, can be constructed hierarchically. A hierarchical RCDF graph can contain one or more *hierarchical actors* (also called super actors), where a hierarchical actor is an actor whose internal functionality is represented by a (nested) RCDF graph. Such a nested RCDF graph can also be hierarchical, so hierarchical RCDF specifications can be arbitrarily "deep." By applying hierarchical organization to Figure 2, we can derive the hierarchical RCDF graph shown in Figure 7. The actor $O$ in the hierarchical graph represents the nested graph of Figure 2.

We apply MVP in the hierarchical graph of Figure 7, and the graph that results from this combination of hierarchy and MVP is shown in Figure 8. The performance improvement associated with this transformed representation is shown in Table IV.
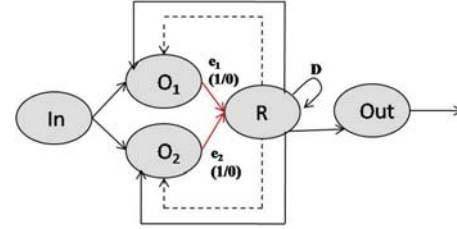
## IV. APPLICATION: CESSNA AIRCRAFT

We take an example due to Maciejowski et al. where MPC is used to control a Cessna citation 500 aircraft [4] when it is cruising at an altitude of 5000m and a speed of 128.2 m/sec. The structure of the implementation of the whole system is



In—InputFunction; O—SolveOptimizationProblem;
Out—OutputResult;

Fig. 7. Hierarchical dataflow representation.



MTCEs: {$e_1$, $e_2$}
In—InputFunction; $O_1$—InteriorPointOption;
$O_2$—ActiveSetOption; Out—OutputResult;

Fig. 8. MVP version of hierarchical RCDF model.

shown in Fig. 9. As shown in Fig. 9, the active set method described above is used to produce the control input.

There is only one input: elevator angle. There are three outputs: pitch angle, altitude and altitude rate.

The following is the constant-speed approximation of linearized dynamics of the aircraft:

$$\dot{x} = Ax + Bu,$$

$$y = Cx,$$

where

$$A = \begin{bmatrix} -1.2822 & 0 & 0.98 & 0 \\ 0 & 0 & 1 & 0 \\ -5.4293 & 0 & -1.8366 & 0 \\ -128.2 & 128.2 & 0 & 0 \end{bmatrix}, B = \begin{bmatrix} -0.3 \\ 0 \\ -17 \\ 0 \end{bmatrix},$$

and

$$C = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ -128.2 & 128.2 & 0 & 0 \end{bmatrix}, D = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}.$$

The elevator angle is limited to $\pm 15^o(\pm 0.262 rad)$, and the elevator slew rate is limited to $\pm 30^o/sec$. These are limits imposed by the equipment design, and cannot be exceeded. For passenger comfort the pitch angle is limited to $\pm 20^o$.

The performance measure for this application is formulated as a QP problem: $f(z) = \frac{1}{2}z^T Q z + c^T z$, and $Jz \leq g$.

TABLE IV

SIMULATION RESULTS OF THE SYSTEM IN HIGHER HIERARCHY

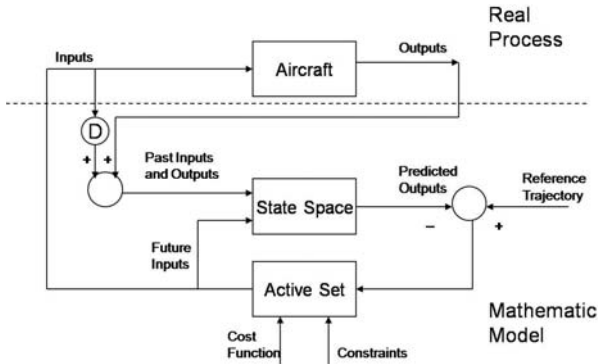| ncond | system without MVP | | system with MVP | |
|---|---|---|---|---|
| statistics | mean | variance | mean | variance |
| 0 | 0.29844 | 0.010 | 0.15937 | 0.000 |
| 3 | 0.29375 | 0.007 | 0.29370 | 0.005 |
| 6 | 0.36594 | 0.003 | 0.36500 | 0.002 |
| 9 | 0.50313 | 0.008 | 0.36406 | 0.003 |
| 12 | 0.53281 | 0.005 | 0.28594 | 0.007 |

Fig. 9. Complete MPC system of aircraft

TABLE V

AVERAGE COMPUTING TIME IN SECONDS FOR AIRCRAFT EXAMPLE.

| $T_s(sec)$ | original | MVP+DPT | improve |
|---|---|---|---|
| 0.5 | 0.10643 | 0.04039 | 62.0% |
| 0.05 | 0.10724 | 0.04872 | 54.5% |

An MPC controller was designed by Maciejowski et al. with the sampling interval $T_s = 0.5s$, the prediction horizon $N_p = 10$, the control horizon $N_u = 3$. The following constraints : $|u| \leq 0.262$, $|\Delta u| \leq 0.524$, $|y_1| \leq 0.349$ were also included. The system has to solve on-line the QP problem at every sampling time.

Table. V shows the time required on the average for one time step assuming each actor is implemented in Matlab and that parallel actors run simultaneously. The second row of Table. V indicates what would happen for a much smaller sampling interval. For this problem, such a short sampling interval would be unwise because it results in a relatively poorly conditioned problem. Even so, if the algorithm using PMV+DPT were programmed into an FPGA, for example, it would run quickly enough to be usable with the faster dynamics and it is not clear that the original implementation could be made fast enough to be usable.

## V. CONCLUSIONS AND SUGGESTIONS FOR FURTHER RESEARCH

In this paper and its predecessor [1], we have demonstrated a set of tools for analyzing an MPC algorithm, identifying bottlenecks, and suggesting ways to reduce the time needed for it to complete its computations. The main point is that given an algorithm that needs to run in real time (not necessarily an MPC algorithm) and a class of problems to be solved, we have tools to improve its performance. These tools identify areas of possible improvement. One still has to find ways to introduce parallelism into the execution of the algorithm.

We have not identified or exploited all the opportunities for parallelism in the two algorithms we have analyzed so far. For example, both require the solution of a large set of linear equations. There is a large literature on how to perform such calculations in parallel. We nonetheless perform those computations sequentially in both of our papers.

It would be desirable to provide both an optimized algorithm for a large class of MPC problems and an efficient, reliable, and fast implementation of that algorithm. The result would be a kind of plug and play MPC. To do this one would need a good set of test candidates for MPC control and a reasonable collection of candidate algorithms. Our tools could then be applied to analyze the algorithms, find ways to parallelize them, and develop special purpose hardware to implement them efficiently and reliably. The resulting MPC controller could greatly extend the applications of MPC in the real world.

## REFERENCES

[1] R. Gu, S. S. Bhattacharyya, and W. S. Levine, "Dataflow-based implementation of model-predictive control," in *Proceedings of American Control Conference*, June 2009.

[2] R. Bartlett, A. Wachter, and L. T. Biegler, "Active set vs. interior point strategies for model predictivecontrol," in *Proceedings of American Control Conferenc*, vol. 6, June 2000, pp. 4229–4233.

[3] L. G. Bleris, J. Garcia, M. G. Arnold, and M. V. Kothare, "Towards embedded model predictive control for system-on-a-chip applications," *Journal of Process Control*, vol. 16, no. 3, March 2006.

[4] J. M. Maciejowski, *Predictive Control with Constraints*. Prentice Hall, 2002.

[5] P. E. Gill, W. Murray, and M. Wright, *Practical Optimization*. Academic Press, London, UK, 1981.

[6] P. A. Absil and A. L. Tits, "Newton-kkt interior-point methods for indefinite quadratic programming," *Computational Optimization and Applications*, vol. 36, no. 1, pp. 5–41, January 2007.