2009 American Control Conference
Hyatt Regency Riverfront, St. Louis, MO, USA
June 10-12, 2009

Dataflow-based Implementation of Model Predictive Control

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

Abstract— **Model Predictive Control (MPC) has been used in a wide range of application areas including chemical engineering, food processing, automotive engineering, aerospace, and metallurgy. MPC is often computation intensive, which limits the class of systems to which it can be applied and the performance criteria it can use. This paper describes a general framework called reactive, control-integrated dataflow modeling for analyzing and improving the algorithms used for MPC and their hardware implementations. The utility of the framework is demonstrated by applying it to the Newton-KKT algorithm. The results show significant reductions in computation time for test cases.**

I. INTRODUCTION

Model Predictive Control (MPC) has found broad application, especially in the process industry. MPC can require considerable amounts of computation because it is based on the real-time solution of an optimal control problem. This has limited its application to relatively slow systems and tractable performance measures [1]. The ability to solve larger, more complex problems quickly would extend the practical applicability of MPC.

As a result, there has been considerable research aimed at speeding up the computation of optimal controls. Most of this research has concentrated on improving the algorithms. Relatively little work [2] 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 paper describes a high level method for modeling control algorithms. The resulting models display the flow of data and the sequencing of calculations in a way that greatly facilitates their analysis. In particular, it is relatively easy to see where computational and/or storage bottlenecks exist. Once identified, these problems can be eliminated or ameliorated by modifying the algorithm or by proper hardware implementation. The method and its use are demonstrated by an example application to the Newton-KKT algorithm, a potentially important component of future MPC algorithms.

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

II. RELATED WORK

A. Control Background

MPC has been studied at least since the 1970s. At that time various works show an incipient interest in MPC in the process industry [3][4]. The basic ideas appearing in MPC are explicit use of a model to predict the process output at future time instants; calculation of a control sequence minimizing a certain objective function; and the application of only the first control signal of the sequence calculated at each step. A detailed introduction to MPC and some specific algorithms can be found in the book [5].

It is well known that MPC can be computation intensive and that, as a result, it can usually be used only in applications with relatively slow dynamics [1]. One approach to addressing this problem has been to compute the control law off-line and store it as a lookup table [6]. However, the situations where this can be done are limited. One would like to be able to compute the controls in real time by solving an optimal control problem. This has prompted a number of researchers to investigate means for increasing the speed with which optimal controls can be computed. Much of this work has focused on improving the algorithms [1], [7].

A few researchers have addressed the implementation of MPC. Ling et al. [8] demonstrated that a "reasonably sized constrained MPC Controller" could be implemented on a modest FPGA chip. Bleris et al. [9] have proposed a computing architecture that is specifically designed for MPC. Furthermore, they have proposed a design framework for application specific processor implementation [2]. Our approach differs from that of Bleris et al. in that we focus on modeling the MPC algorithm structure. This model can be used to derive efficient implementations across a range of architectures. In particular, designers can systematically trade off performance and resource requirements, based on the constraints of the control problem, and the set of available hardware resources.

B. Embedded Signal Processing Background

Since the mid 1980s, a class of graphical program representations has been evolving steadily, and gaining increasing acceptance among designers of digital signal processing (DSP) systems. Foundations for such *dataflow* representations have been provided by computation graphs [10], Kahn process networks [11], and dataflow architectures [12]. Synchronous dataflow (SDF) is a specialized form of dataflow that is streamlined for efficient representation of DSP systems [13]. Since the introduction of SDF, a variety of such

978-1-4244-4524-0/09/$25.00 ©2009 AACC

DSP-oriented dataflow models of computation have been proposed, and DSP-oriented models have been incorporated into many commercial design tools, including Agilent ADS, Cadence SPW (later acquired by CoWare), National Instruments LabVIEW, and Synopsys CoCentric. These alternative modeling approaches provide different trade-offs among expressive power (the range of DSP applications that can be represented), analysis potential (the rigor with which implementations can be automatically validated or optimized), and intuitive appeal.

In 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 pre-defined 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).

A limitation of SDF and related models, such as cyclo-static [14] dataflow, is that dynamic dataflow relationships among computations cannot be described. To express applications that involve such relationships, one must employ models that are more expressive than such *static dataflow* models. Earlier work on DSP-oriented dataflow models has focused heavily on static dataflow techniques, especially SDF. As designers seek to develop more and more complex embedded DSP systems, incorporating more flexible sets of features, and more powerful forms of adaptivity, exploration of dynamic dataflow models is becoming increasingly important.

A variety of dynamic dataflow modeling techniques have been developed previously, including stream-based functions [15], functional DIF [16], and the CAL actor language [17]. In this paper, we describe a new dynamic dataflow modeling technique, called reactive, control-integrated dataflow (RCDF), that appears particularly promising for MPC applications. Our approach is more specialized compared to other dynamic dataflow techniques, but for MPC, this specialization can be exploited in useful ways to streamline the implementation process.

Note that in addition to their formal properties, DSP-oriented dataflow models provide different kinds of software architectures for working with signal processing computations (of which control system implementations form an important sub-class). This kind of representation can help to structure subsequent phases of design, simulation, verification, testing, and implementation *regardless of whether the underlying model of computation is explicitly supported by an off-the-shelf design tool*. This is true especially in the area of embedded systems, including embedded control, where designers are often willing to explore specialized, application/architecture-driven analysis techniques that may provide streamlined performance, power consumption, cost, or robustness.

III. REACTIVE CONTROL-INTEGRATED DATAFLOW

We first introduce some notation related to dataflow graphs. Given an edge e in a dataflow graph, we denote the source and sink actors of e as $src(e)$ and $snk(e)$, respectively. For a specific edge e_i, and a positive integer j, we use $prd(e_i, j)$ to denote the number of tokens produced by $src(e_i)$ onto e_i during the jth execution of $src(e_i)$, and similarly, we use $cns(e_i, j)$ to denote the number of tokens consumed by $snk(e_i)$ from e_i during the jth execution of $snk(e_i)$. In general, $prd(e_i, j)$ and $cns(e_i, j)$ can be data dependent — i.e., they can depend on the values of samples in the input signals of the dataflow graph. In the restricted case of SDF, such data dependence cannot be present, and furthermore, there can also be no dependence on j — that is, the production and consumption "volumes" must be the same for all values of j.

Given a dataflow graph edge e and a positive integer k, we say that e has *k/0 production* if for all j, $prd(e, j) \in \{0, k\}$. Similarly, a dataflow edge has *k/0 consumption* if for all j, $cns(e, j) \in \{0, k\}$. Notice that by this definition, any edge e in an SDF graph (degenerately) has k'/0 consumption for some $k' = k'(e)$

If S is a set of k/0 production (consumption) edges in a dataflow graph, we employ a minor abuse of notation and refer to S as having k/0 production (consumption).

A major concept in the RCDF modeling approach is that of *mutually-exclusive* production and consumption edges. This concept provides a common framework for representing a useful class of dynamic dataflow structures that is relevant to MPC.

Definition 1: Given a dataflow graph G, a set of *mutually-exclusive token-production edges* (*METP* edges) is a subset $e_1, e_2, \ldots, e_m$ of edges in G such that for any set of input signals applied to G, and for any positive integer j,

$$\sum_{i=1}^{m} I(prd(e_i, j)) = 1, \text{ where } I(x) = \begin{cases} 1, x > 0 \\ 0, x \leq 0 \end{cases}$$

represents the indicator function over the positive integers.

Intuitively, a set S of edges is an METP set if across any set of corresponding executions of the edges' source actors, a nonzero token volume is produced on exactly one of the edges in S.

An analogous notion of *mutually-exclusive token-consumption edges* (*METC* edges) can be formulated by replacing the sum in Definition 1 with

$$\sum_{i=1}^{m} I(cns(e_i, j)) = 1,$$

.

In this paper, we focus on a particular class of METP edges and METC edges, which we refer to as *regular* METP edges and METC edges. A regular METP is an METP $S = e_1, e_2, \ldots, e_m$ such that all elements have the same source actor ($src(e_i) = src(e_1)$ for all i), and there exists a positive integer k such that S has k/0 production. Similarly, a regular METC is an METC such that all elements have the same sink actor, and there exists a positive integer k such that the METC has k/0 consumption.

IV. EXAMPLE: NEWTON KKT ALGORITHM

In numerical analysis, Newton's method is one of the best known methods to find successively better approximations to the roots of a real-valued function. Newton's method, as an optimization algorithm, is also well-known for finding the value of $x \in R^n$ that minimizes a twice-differentiable function $f : R^n \rightarrow R$.

The Karush-Kuhn-Tucker conditions (KKT) are necessary for a solution of a nonlinear programming problem to be optimal provided some regularity conditions are satisfied.

The Newton-KKT method takes advantage of both Newton's method and the KKT condition. Newton-KKT methods are algorithms in which search directions for the primal variables and the KKT multiplier estimates are components of the Newton (or quasi-Newton) direction for the solution of the equalities in the first-order KKT conditions of optimality or a perturbed version of these conditions. The specific version of Newton KKT we use was introduced by Absil and Tits [18]. Their methods are adapted from previously proposed algorithms for convex quadratic programming and general nonlinear programming. The Newton-KKT algorithm is a good choice for use in solving the discrete-time optimal control problems that are central to MPC.

Consider the following Quadratic Programming (QP) problem in standard inequality form:

$$\text{minimize } \frac{1}{2} < x, Qx > + < c, x >, \text{ s.t. } Ax \leq b, x \in R^n$$

with $A \in R^{m \times n}$, $b \in R^m$, $c \in R^n$, and with $Q \in R^{n \times n}$ symmetric. In order to solve this problem using the Newton KKT method, we introduce some notation. Let $I = 1, ...m$, where m is the number of rows of A, and, for $i \in I$, let a_i be the ith row of A, let b_i be the ith entry of b, and let $g_i(x) \equiv < a_i, x > -b_i$. Also let $f(x) \equiv \frac{1}{2} < x, Qx > + < c, x >$. The Newton KKT algorithm to solve the problem is modeled by the RCDF model in Figure 1. We implement communication between actors based on the dataflow model. However implementation of each actor follows the sequential programming method. As shown in Figure 1, there are seven actors in the system, and each actor is responsible for a specific function. The function of each actor is described in brief as follows:

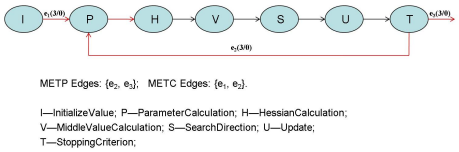

Fig. 1. RCDF model of Newton KKT algorithm.

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.

P—The actor P is used to compute the values of f, g and the Schur complement at the current value of x for every iteration.

H—The actor H is used to compute the modified Hessian matrix. It functions only under the condition that the Hessian matrix has one or more negative eigenvalues.

V—The actor V is used to compute the gradient of f in every iteration.

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.

U—The actor U is used to compute the updated values of x, f and g.

T—The actor T is used to compare the difference between the updated value and previous value with a given criterion, to see if the system needs to go to the next iteration or terminate in this iteration.

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 U, may represent *hierarchical actors* whose internal functionalities are described by additional ("nested") dataflow graphs. We elaborate on the internal representation of actor U in the following section.

V. PROFILING AND IDENTIFICATION OF BOTTLENECKS

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. In our analysis and use of execution time information, we have ignored certain "fine-grained" actors that have very low computational cost. For example, we have ignored actor I, which is used only to initialize parameters. We have also ignored the execution time contributions of actors P, V and T, which involve simple addition and equality-testing operations.

In our MATLAB simulations, the matrix Q was generated based on the condition number ($ncond$) parameter, and

TABLE I
ACTOR EXECUTION TIMES (SEC) IN NEWTON-KKT.

$ncond$	0		3		6	
statistics	mean	variance	mean	variance	mean	variance
H	0.012	4e-5	0.011	5e-5	0.012	4e-5
S	0.009	6e-5	0.006	5e-5	0.026	7e-5
U	0.005	5e-5	0.005	5e-5	0.010	5e-5
V	0.000	0.000	0.000	0.000	0.000	0.000

the number of negative eigenvalues ($ngeig$). We fixed the parameter $ngeig$ to be 5, and chose $ncond$ from the set $\{0, 3, 6, 9, 12\}$. Table I shows the execution times of different actors for different values of $ncond$ and otherwise randomly chosen Q. These values were determined by implementing each actor in MATLAB (version 7.04), executing the code for 10 random choices of Q, and recording the mean and variance of the time required to complete the computations.

It is easily seen from Table I that actors H, S, and U impose a relatively high computational load.

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.

VI. MULTI-VERSION PARALLELISM

Two important forms of parallelism that are exposed effectively by dataflow graphs are functional parallelism and data 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*, 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. As an example of multi-version parallelism, consider the search direction calculation actor in the NEWTON-KKT example of Section IV. Two different algorithms are given in [18] for determining the search direction. The first, which we denote by actor S_1, is an augmented system. In some applications, a normal system, also given in [18] converges faster. We denote it 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 jth execution of S_1 will complete before the jth execution of S_2, and for other values $j \in J_2$ ($J_1 \cap J_2 = \emptyset$), the jth 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.

If one replaces an actor X with actors $x_1, x_2, \ldots, x_n$ that represent multiple versions of X, then with respect to the new (transformed) dataflow graph, parallelism among $x_1, x_2, \ldots, x_n$ can be viewed as functional parallelism. Multi-version implementation is also related to data parallelism since the parallel executions of $x_1, x_2, \ldots, x_n$ operate on one or more common data streams in the original dataflow graph (the data streams associated with the input edges of X). Thus, in some sense, we can view multi-version parallelism as a hybrid form of parallelism that involves aspects of both functional and data parallelism.

VII. CASE STUDY: NEWTON-KKT

Through our execution time analysis on our model of Newton-KKT, we found, as described in Section V, that the major computational bottlenecks are the actors H, S, and U.

A. Multi-version implementation of H

To alleviate the bottleneck due to H (Hessian calculation), we apply a multi-version implementation of H. Two different methods for adjusting the Hessian to be positive definite are mentioned in [18]. The first, H_1, is more reliable while the second, H_2, is usually faster. The new dataflow graph that results from applying the multi-version transformation to H and S is illustrated in Fig. 2. Here, H_1 and H_2 represent the computation-every-iteration and vector-based-computation methods, respectively.

Actor R in Figure 2 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

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. These signals are asynchronous in the sense that they are not synchronized with dataflow firings (executions) of the actors that they are controlling. Such reset signals are useful, for example, to save execution time and power consumption associated with version executions that are ignored because another version has "won the race" already for the current execution. Such a signal can be implemented through a software interrupt or through asynchronous hardware reset logic, depending on the type of implementation platform. The asynchronous reset signals generated by the MVOS actor R are represented by dashed lines in Figure 2. The main drawback associated with using these reset signals is that they deviate from pure dataflow semantics, and this may complicate certain forms of analysis of the overall specification. Studying ways to systematically integrate such asynchronous reset signals into DSP-oriented dataflow graph analysis is a useful direction for further study.

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.

B. Handling failed searches

Multi-version implementation is especially attractive for scenarios in which complex searches must be carried out. In practice, such search techniques can sometimes fail to find solutions within the allowable period of time that can elapse before a response must be produced by the system. In such cases, if the system keeps waiting, the whole system may stop or "crash", leading to disastrous or otherwise undesirable consequences. This is important in MPC because the optimization computation may fail to converge in the available time.

To prevent such failures, timers can be incorporated into MVOS actors so that whenever a timeout occurs, asynchronous reset signals, and next-execution-enable tokens are sent to all of the associated versions. In such cases, the MVOS actor can respond with a copy of the value that it produced by the previous execution of the corresponding set of version actors. More elaborate approaches for handling timeout problems in multi-version actors are worthy of further investigation.

C. Transformation of U

To alleviate the bottleneck due to U, we examined the MATLAB source code for U and replaced it with the equivalent, hierarchical ("nested") RCDF graph shown in Figure 3. We performed this MATLAB-to-RCDF transformation manually, through our understanding of the algorithm and relationships among its various parts. Because of the high expressive power of MATLAB, automated conversion of MATLAB to specialized DSP-oriented dataflow representations is in general undecidable (computationally infeasible), although investigation of such conversion under restricted cases is an interesting direction for further study.

Our refinement of actor U as a nested RCDF graph exposes opportunities for exploiting functional parallelism among actors U_{t2}, U_{t3}, and U_{t4}. Also, the U_{t1} and U_{t2} actors are both well-suited to exploiting data parallelism because they involve relatively simple operations that are applied independently to successive items of data in their respective input streams. In our experiments, we have exploited both the functional parallelism and data parallelism described above that is associated with our RCDF refinement of actor U. These experiments are described in Section VII-D.

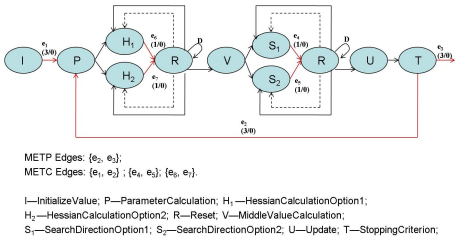

Fig. 2. RCDF model of Newton KKT algorithm after application of multi-version transformations.

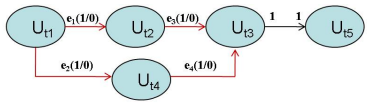

Fig. 3. RCDF model of Newton KKT algorithm after transformations.

D. Experimental Results

To demonstrate our parallel processing methods for Newton KKT, we have evaluated them with MATLAB simulations. These simulations take into account the detailed functionality of each actor, and our analysis of the simulation results provides estimates for the performance improvements gained through our integrated application of RCDF modeling, multi-version transformations, and hierarchical refinement.

Our simulations are organized into four groups. In group 1, three different hardware designs are simulated.

1) case1(a): sequential system with actor S_1;
2) case1(b): sequential system with actor S_2;

TABLE II

SIMULATION RESULTS FOR GROUP 1.

$ncond$	Case1(a)		Case1(b)		Case2	
statistics	mean	variance	mean	variance	mean	variance
0	0.309	0.031	0.054	0.000	0.055	0.000
3	0.758	0.013	1.452	0.005	0.758	0.013
6	0.839	0.015	0.530	0.002	0.525	0.003
9	0.769	0.014	0.330	0.003	0.330	0.002
12	1.173	0.018	0.334	0.012	0.334	0.010

TABLE III

SIMULATION RESULTS FOR GROUP 2.

$nzeig$	Case3(a)		Case3(b)		Case4	
statistics	mean	variance	mean	variance	mean	variance
function 1	1.778	0.007	2.031	0.010	1.714	0.007
function 2	1.875	0.003	2.040	0.005	1.859	0.003
function 3	1.869	002	1.967	0.006	1.809	0.003
function 4	2.834	0.001	2.888	0.012	2.803	0.012
function 5	3.422	0.001	3.433	0.016	3.260	0.011

3) case2: parallel, multi-version system using both S_1 and S_2.

In Table II, different values $(0, 3, 6, 9, 12)$ for the parameter $ncond$ determine different objective functions that are input to the Newton KKT system. From the simulation results, the parallel, multi-version architecture outperforms both of the sequential architectures. This is because different versions (S_1 or S_2) perform better for different problems.

The second group of simulation, group 2, is as follows:

1) case3(a): sequential system with actor H_1;
2) case3(b): sequential system with actor H_2;
3) case4: parallel, multi-version system using both H_1 and H_2.

In this case, $ncond$ was held fixed, and $ngeig$ was varied because H only activates when $ngeig > 0$. The simulated results share similar properties with the results of group 1 — the multi-version architecture again provides significant performance improvement.

Next, in group 3, we examine the effect of combining both of the instances of multi-version parallelism that we are experimenting with.

1) case4: parallel system with multi-version implementation of H (H_1 and H_2), and single-version implementation of S (S_1 only).
2) case5: parallel system with multi-version implementation of S (S_1 and S_2), and single-version implementation of H (H_1 only).
3) case6: parallel system with multi-version implementations of both H and S.

The simulation results for group 3 are shown in Table IV. In these results, it is demonstrated that the more multi-version parallelism we utilize, the better the system performance.

TABLE IV

SIMULATION RESULTS FOR GROUP 3.

$ncond$	Case4		Case5		Case6	
statistics	mean	variance	mean	variance	mean	variance
0	2.611	0.011	0.164	0.000	0.128	0.001
3	2.175	0.004	0.541	0.002	0.532	0.000
6	3.275	0.012	0.715	0.005	0.713	0.002
9	2.784	0.008	0.766	0.003	0.756	0.006
12	4.466	0.010	1.117	0.003	1.084	0.008

TABLE V

SIMULATION RESULTS FOR GROUP 4.

$ncond$	Case7		Case8	
statistics	mean	variance	mean	variance
0	0.794	0.009	0.725	0.008
3	1.111	0.010	1.067	0.012
6	1.206	0.012	1.110	0.013
9	0.991	0.015	0.904	0.015
12	1.633	0.018	1.531	0.016

The simulations in groups 1-3 involve functional parallelism that is achieved through the multi-version transformation. Next, we examine the effect of data parallelism. We fix the value of $ngeig$ at 0 , and vary the $ncond$ parameter. Under this condition, all of the problems are convex and neither H_1 nor H_2 is needed. We summarize the fourth group of experiments as follows.

1) case7: sequential system with the original version of U;
2) case8: parallel system using the data parallel version of U described in Section VII-C.

The simulation results for group 4 are shown in Table V. These results show that data parallelism also gives a significant improvement in system performance.

The results in groups 1-4 help to understand the impact of individual dataflow graph transformations in isolation. In our next group of experiments, we show the impact of applying all of the transformations together. The results in Table VI compare the performance of a sequential implementation with that of a "fully-transformed implementation — that is, and implementation that includes multi-version implementations of both H and S , as well as a data parallel implementation of U.

To further demonstrate the utility of these tools we took an example due to Maciejowski et al. where MPC is used to control a Cessna citation 500 aircraft [19] when it is cruising at an altitude of 5000m and a speed of 128.2 m/sec. They use a linear fourth-order model of the aircraft. There is only one input: elevator angle. There are three outputs: pitch angle, altitude and altitude rate.

The elevator angle is limited to $\pm 15^o$, and the elevator

TABLE VI

SIMULATION RESULTS FOR GROUP 5.

$ncond$	Case1(a)		Case9		improve
statistics	mean	variance	mean	variance	percentage
0	2.735	0.008	0.161	0.000	94.1%
3	2.310	0.012	0.517	0.001	77.6%
6	3.040	0.007	0.636	0.000	79.1%
9	2.900	0.007	0.816	0.005	81.6%
12	4.363	0.016	1.061	0.002	75.7%

TABLE VII

AVERAGE COMPUTING TIME IN SECONDS FOR AIRCRAFT EXAMPLE

$T_s(sec)$	original	PM+DPV	improve
0.5	0.0859375	0.0281250	67.2%

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 is $f(z) = \frac{1}{2}z^TQz + c^Tz$. An MPC controller was designed by Maciejowski et al. with $0.5s$ sampling interval, prediction horizon $N_p = 10$, and control horizon $N_u = 3$. The system must compute the solution to the Quadratic Programming obtained by time discretization of this continuous-time optimal control problem at every sample time and within the sampling interval.

We simulated implementations of the Maciejowski et al. MPC using the Newton-KKT algorithm described in Section VII. Table VII shows the average time required per time step for the computations to finish. Note that the original scheme in the table refers to the Newton-KKT, not the Maciejowski et al. algorithm. Clearly, we have been able to shorten the time required for the computations substantially. The extra computing time created could be used to improve the MPC controller in a variety of ways.

VIII. CONCLUSIONS

In this paper, we have proposed an abstract model for the control algorithms used in MPC, and have analyzed the model for performance bottlenecks and, in examples, possible improvements. We have introduced different forms of parallelism into the algorithm, and demonstrated the resulting improvements in performance. Our approach to parallel MPC implementation can also increase system reliability in the following sense. The calculation does not necessarily terminate with an optimal control. It is known that MPC will be stable provided the control obtained is, loosely speaking, close enough to optimal. Parallelism increases the likelihood that this is so.

Notice that the tools described in the paper are largely implementation independent. The methodologies for exploiting functional, multi-version, and data parallelism described in the paper have broad, and so far only partially exploited applicability to MPC and other control system computations.

REFERENCES

[1] Y. Wang and S. Boyd, "Fast model predictive control using online optimization," in *Proceedings of the IFAC World Congress*, July 2008, pp. 6974–6979.

[2] 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.

[3] J. Richalet, A. Rault, J. L. Testud, and J. Papon, "Model predictive heuristic control: application to industrial processes," *Automatica*, vol. 14, no. 2, pp. 413–428, 1978.

[4] C. R. Cutler and B. Ramaker, "Dynamic matrix control—a computer control algorithm," in *Proceedings of the Joint Automatic Control Conference*, 1980.

[5] E. F. Camacho and C. Bordons, *Model predictive control in the process industry*. Springer, 1995.

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

[7] H. Chung, E. Polak, and S. Sastry, "An accelerator for packages solving discrete-time optimal control problems," in *Proceedings of the IFAC World Congress*, July 2008, pp. 14 295–14 300.

[8] K. V. Ling, S. P. Yue, and J. M. Maciejowski, "An FPGA implementation of model predictive control," in *Proceedings of the American Control Conference*, June 2006.

[9] L. G. Bleris, P. D. Vouzis, M. G. Arnold, and M. V. Kothare, "A co-processor FPGA platform for the implementation of real-time model predictive control," in *Proceedings of the American Control Conference*, June 2006.

[10] R. M. Karp and R. E. Miller, "Properties of a model for parallel computations: Determinacy, termination, queuing," *SIAM Journal of Applied Math*, vol. 14, no. 6, November 1966.

[11] G. Kahn, "The semantics of a simple language for parallel programming," in *Proceedings of the IFIP Congress*, 1974.

[12] J. B. Dennis, "First version of a data flow procedure language," Laboratory for Computer Science, Massachusetts Institute of Technology, Tech. Rep., May 1975.

[13] E. A. Lee and D. G. Messerschmitt, "Synchronous dataflow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, September 1987.

[14] G. Bilsen, M. Engels, R. Lauwereins, and J. A. Peperstraete, "Cyclo-static dataflow," *IEEE Transactions on Signal Processing*, vol. 44, no. 2, pp. 397–408, February 1996.

[15] B. Kienhuis and E. F. Deprettere, "Modeling stream-based applications using the SBF model of computation," in *Proceedings of the IEEE Workshop on Signal Processing Systems*, September 2001, pp. 385–394.

[16] W. Plishker, N. Sane, M. Kiemb, K. Anand, and S. S. Bhattacharyya, "Functional DIF for rapid prototyping," in *Proceedings of the International Symposium on Rapid System Prototyping*, Monterey, California, June 2008, pp. 17–23.

[17] J. Eker and J. W. Janneck, "CAL language report, language version 1.0 — document edition 1," Electronics Research Laboratory, University of California at Berkeley, Tech. Rep. UCB/ERL M03/48, December 2003.

[18] 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.

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